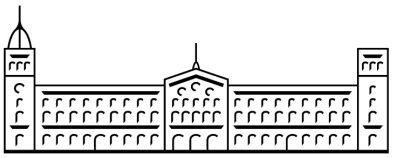

Treball final de grau

GRAU DE MATEMÀTIQUES

Facultat de Matemàtiques
Universitat de Barcelona

Automated Theorem Proving

Autor: Miquel Montserrat Armstrong

Director: Dr. Juan Carlos Martínez
Realitzat a: Departament de Matemàtiques i Informàtica

Barcelona, June 27, 2016

Abstract

Automated Theorem Proving is an area in mathematical logic and computer science dedicated to the production of theorem proofs by algorithmical means, and is of great use in many fields such as some areas of mathematics, artificial intelligence, software verification, hardware verification or declarative programming.

The aim of the first part of this work is to present Herbrand's theory and the Resolution Method for the first order logic. In the second part we show one of the many applications of the Resolution Method: the Prolog programming language, which is a useful tool for the resolution of problems in the NP class.

Contents

1 Introduction

The interest of man in finding a general decision procedure to determine if a given mathematical assertion is true or false is an old matter. Back in the seventeenth century, Gottfried Leibnitz dreamt of building a machine that could manipulate symbols with the purpose of finding the truth value of any given mathematical assertion, this is, if it was true or false.

In 1928 David Hilbert posed the decision problem, known also as the *Entscheidungsproblem*, which asks for a general algorithm that determines if a given formula in first order logic is a *valid* formula.

In 1930 Jacques Herbrand proposed the first mechanical method to prove theorems, this is, to prove validity of first order formulas. Though his method is far too time consuming to be carried out even by modern computer systems, his approach was a big improvement.

A proof that the validity question in first order logic is undecidable, implying a negative answer to the Entscheidungsproblem, was given in 1936 by Alonzo Church. This result implies the fact that, given a valid formula, to verify its validity algorithmically is the most that can be done.

In this sense, after Gilmore's implementation of Herbrand's procedure on a digital computer in 1960, a major breakthrough was made by J.A. Robinson in 1965 with the development of the Resolution Method. This procedure and its subsequent refinements have become of great significance in Automated Theorem Proving and its many applications. One of the latter is the Prolog programming language which, because of its features, is of great use for finding solutions for a variety of practical problems in the NP class.

Automated Theorem Proving deals with the search, by means of some algorithm, for theorem proofs. For this, reasoning and inference processes are formalized in some kind of symbolic logic, typically first order logic, and the matter consists in showing that the statement made by the theorem is a *logical consequence* of the hypotheses.

2 Fundamentals of Logic

Symbolic logic considers languages whose essential purpose is to symbolize reasoning encountered not only in mathematics, but also in daily life.

The aim of this section is to introduce necessary basic concepts for Automated Theorem Proving.

2.1 The Propositional Logic

In the propositional logic, the matter of interest are declarative sentences that can be either true or false. Such declarative sentences are called *propositions*.

Definition 2.1. An *atom* is a proposition that can not be broken down into more simple propositions.

In the propositional logic five logical connectives are used: $\neg, \wedge, \vee, \rightarrow$, and $\leftrightarrow$ with their usual meaning and truth tables. Next, the propositional language can be defined.

Definition 2.2. (Syntax of propositional logic) If σ is a countable set of atoms, we define the *language of the σ-propositional formulas* as the set of elements that are generated by the following rules:

i. Every atom of σ is a σ-formula.

ii. If ϕ is a σ-formula, then $(\neg\phi)$ is also a σ-formula.

iii. If $\phi_1, \ldots, \phi_n$ are σ-formulas, then $(\phi_1 \vee \ldots \vee \phi_n)$ and $(\phi_1 \wedge \ldots \wedge \phi_n)$ are also σ-formulas.

iv. If ϕ and ψ are σ-formulas, then $(\phi \rightarrow \psi)$ and $(\phi \leftrightarrow \psi)$ are also σ-formulas.

Remark 2.3. Often parentheses are suppressed to keep formulas readable if this results in no ambiguity.

Definition 2.4. (Semantics of propositional logic) If σ is a countable set of atoms, a *σ-interpretation* is a mapping I: $\sigma \longrightarrow \{T, F\}$, where T denotes the truth value TRUE and F denotes the truth value FALSE.

Having defined the notion of interpretation, we can evaluate a formula ϕ under a certain interpretation I by replacing each atom A_i in ϕ by its truth value $I(A_i)$, now the truth value of the formula is found by using the usual truth tables for the different connectives.

Example 2.5. Let $\sigma = \{P, Q, R\}$. We define the interpretation I as:

I(P)=T, I(Q)=F, I(R)=F.

Let $\phi = (P \to Q) \vee R$, then we have

$I(\phi) = (T \to F) \vee F = F \vee F = F.$

Now we define further basic notions for the propositional logic.

Definition 2.6. A formula ϕ is called *satisfiable* or *consistent* if there exists an interpretation I such that $I(\phi) = T$. In this case we say I is a *model* for ϕ. ϕ is called a *valid formula* or *tautology* if $I(\phi) = T$ for any interpretation I, and if ϕ is false under all interpretations it is said to be *inconsistent* or *unsatisfiable*.

Definition 2.7. A set Φ of formulas is *satisfiable* if there exists an interpretation I such that $I(\phi) = T$ for every $\phi \in \Phi$.

Definition 2.8. Two σ-formulas ϕ and ψ are said to be *equivalent* if for all interpretation I $I(\phi) = I(\psi)$.

Definition 2.9. A formula ϕ is a *logical consequence* of formulas $\psi_1, \ldots, \psi_n$ if and only if $\psi_1 \wedge \ldots \wedge \psi_n \to \phi$ is a valid formula, or equivalently $\psi_1 \wedge \ldots \wedge \psi_n \wedge \neg\phi$ is unsatisfiable.

Remark 2.10. Since the number of possible interpretations for a given propositional formula is always finite, for determining if it is a valid formula it is sufficient to evaluate it in all interpretations.

Definition 2.11.

i) A *literal*, also called *elementary formula*, is an atom or the negation of an atom.

ii) A *clause* is a disjunction of literals.

iii) A formula ϕ is said to be in a *Conjunctive Normal Form (CNF)* if and only if ϕ has the form $\phi_1 \wedge \ldots \wedge \phi_n$, where each ϕ_i (i=1,...,n) is a clause.

iv) A formula ϕ in CNF is a *Horn Formula* if every disjunction in ϕ contains at most one positive literal.

Now an important theorem is given, as it will play an important role in the proof of Herbrand's Theorem.

Theorem 2.12. ***(Compactness Theorem)*** *Let σ be a countable set of atoms. A set Φ of σ-propositional formulas is satisfiable if and only if every finite subset of Φ is satisfiable.*

Proof. $\Rightarrow$) If Φ is satisfiable then there exists an interpretation I such that $I(\phi)$=T for every formula $\phi \in \Phi$, hence for any finite subset $\Phi' \subseteq \Phi$ we have that $I(\phi')$=T for every $\phi' \in \Phi'$, therefore Φ' is satisfiable.

$\Leftarrow$) For the purposes of this proof a set of formulas Ψ shall be called finitely consistent if every finite subset of Ψ is consistent. So what is to be proved is that every finitely consistent set is indeed consistent.

We shall call a set of formulas Ψ maximal finitely consistent if Ψ is finitely consistent and for every formula ϕ, either $\phi \in \Psi$ or $\neg\phi \in \Psi$.

Notice that a natural correspondence arises between interpretations and maximal finitely consistent sets. To any interpretation I we can assign the set $\Psi_I = \{\phi : I(\phi) = T\}$. This set is clearly maximal finitely consistent, as for every formula ϕ either ϕ or $\neg\phi$ is true under I and thus is in Ψ_I. Conversely, every maximal finitely consistent set Ψ is in correspondence with the interpretation I_Ψ defined by: $I_\Psi(\phi) = T$ if $\phi \in \Psi$, $I_\Psi(\phi) = F$ if $\phi \notin \Psi$ for every atom ϕ.

Let us now consider a maximal finitely consistent set Ψ and its corresponding interpretation I. Now the following facts are an immediate consequence of Ψ being maximal finitely consistent, and imply (by induction on formula structure) that $\Psi = \Psi_I$:

(1) $\phi \in \Psi$ iff $(\neg\phi) \notin \Psi$

(2) $(\phi \wedge \psi) \in \Psi$ iff $(\phi \in \Psi)$ and $(\psi \in \Psi)$

(3) $(\phi \vee \psi) \in \Psi$ iff $(\phi \in \Psi)$ or $(\psi \in \Psi)$

(4) $(\phi \rightarrow \psi) \in \Psi$ iff $(\phi \notin \Psi)$ or $(\psi \in \Psi)$

For example, let's prove that $(\phi \vee \psi) \in \Psi$ implies $\phi \in \Psi$ or $\psi \in \Psi$. Suppose not. Then $(\phi \vee \psi) \in \Psi$ but $(\neg\phi) \in \Psi$ and $(\neg\psi) \in \Psi$ by maximality. But in this case $\{(\phi \vee \psi), \neg\phi, \neg\psi\}$ is a finite inconsistent set of Ψ, so we would have a contradiction.

By remarks (1)-(4) we can show that proving the finitely consistent set Φ consistent is equivalent to finding a maximal finitely consistent set Ψ such that $\Phi \subseteq \Psi$.

$\Leftarrow$) Let us suppose that there exists a maximal finitely consistent set Ψ such that $\Phi \subseteq \Psi$. Since $I_\Psi \models \Psi$, we deduce that $I_\Psi \models \Phi$, therefore Φ is satisfiable.

$\Rightarrow$) Conversely, supposing Φ consistent, it has to have a model I. I yields a maximal finitely consistent set Ψ_I in the manner formerly described, and as for every $\phi \in \Phi$ $I(\phi) = T$ we have $\Phi \subseteq \Psi_I$.

The way in which to construct such a set Ψ is now shown. Let σ' be the set of atoms that occur in Φ. Since σ' is countable, so is the set of formulas in which only atoms in σ' occur, so they can be enumerated: $\phi_1, \phi_2, \ldots, \phi_n, \ldots$. Now we define $\Phi_0 \subseteq \Phi_1 \subseteq \ldots \subseteq \Phi_n \subseteq \ldots$ as follows:

$\Phi_0 = \Phi$ and

$\Phi_{n+1} = \Phi_n \cup \{\phi_n\}$ if this is finitely consistent

$\Phi_{n+1} = \Phi_n \cup \{\neg\phi_n\}$ otherwise.

Now let $\Psi = \bigcup_{n\geq 0} \Phi_n$. Obviously $\Phi \subseteq \Psi$ and for every formula ϕ, either $\phi \in \Psi$ or $(\neg\phi) \in \Psi$. To finish the proof we need only to show that each Φ_n, and hence Ψ, is finitely consistent. This is proved by induction on n:

- If n=0 we are in the case of the hypothesis of the theorem: Φ is finitely consistent.
- Now, assuming Φ_n is finitely consistent, we prove Φ_{n+1} is finitely consistent:

  -Case 1: $\Phi_{n+1} = \Phi_n \cup \{\phi_n\}$.

  It follows that Φ_{n+1} is finitely consistent by definition.

  -Case 2: $\Phi_{n+1} = \Phi_n \cup \{(\neg\phi_n)\}$.

  By the definition of Φ_{n+1}, $\Phi_n \cup \{\phi_n\}$ is not finitely consistent. Therefore there is some finite set $\Phi'_n \subset \Phi_n$ such that $\Phi'_n \cup \{\phi_n\}$ is not consistent.

  To prove that Φ_{n+1} is finitely consistent, suppose it is not . Then there is a finite set $\Phi''_n \subseteq \Phi_n$ such that $\Phi''_n \cup \{\neg\phi_n\}$ is not consistent. But then $\Phi'_n \cup \Phi''_n$ is a finite subset of Φ_n, and by hypothesis it is consistent.

  Any interpretation I making all formulas of $\Phi'_n \cup \Phi''_n$ true must make either ϕ_n or $\neg\phi_n$ true, contradicting the inconsistency of both $\Phi'_n \cup \{\phi_n\}$ and $\Phi''_n \cup \{\neg\phi_n\}$.

  Thus, in either case, Φ_{n+1} is after all a finitely consistent set.

#

2.2 The Predicate Logic

Predicate Logic, also called First Order Logic, can be understood as an extension of propositional logic, including additional concepts as quantifiers, function symbols and predicate symbols. These new notions allow us to formalize assertions which can not be expressed with the available tools of propositional logic.

In the Predicate Language the following symbols are used:

- Variables
- Constant symbols
- Function symbols
- Predicate symbols
- Logical connectives $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$
- Quantifiers $\exists$ (meaning there exists) and $\forall$ (meaning for all)
- Auxiliary symbols (,) and '

We denote by V the set of variables, and its members are denoted by v_i, i=1,2,... We can assume V, the set of constant symbols, the set of function symbols and the set of predicate symbols are infinite sets.

Function and predicate symbols are related to a positive integer called the *arity* of the symbol, and it corresponds to the number of arguments taken by the symbol. Constant symbols can be considered as 0-arity function symbols.

We shall call any finite set of constant symbols, function symbols and predicate symbols a *vocabulary*. The concept of Predicate Language is defined with respect to a vocabulary, but before doing so the next definition must be given .

Definition 2.13. *Terms* are defined recursively as follows:

i. A constant is a term.

ii. A variable is a term.

iii. If f is an n-place function symbol and $t_1, \ldots, t_n$ are terms, then f$(t_1, \ldots, t_n)$ is a term.

iv. All terms are generated by the above rules.

Definition 2.14. (Syntax of predicate logic) We define now the notion of *Predicate Language*. Let σ be a vocabulary. The set of σ-formulas is defined as the set generated by the following rules:

i. If $R \in \sigma$ is a predicate symbol taking n arguments, and $t_1, \ldots, t_n$ are σ-terms, then R$(t_1, \ldots, t_n)$ is a σ-formula.

ii. If ϕ is a σ-formula, then $(\neg\phi)$ is also a σ-formula.

iii. If $\phi_1, \ldots, \phi_n$ are σ-formulas, then $(\phi_1 \vee \ldots \vee \phi_n)$ and $(\phi_1 \wedge \ldots \wedge \phi_n)$ are also σ-formulas.

iv. If ϕ and ψ are σ-formulas, then $(\phi \rightarrow \psi)$ and $(\phi \leftrightarrow \psi)$ are also σ-formulas.

v. If ϕ is a σ-formula and x is a variable, then $(\exists x\phi)$ and $(\forall x\phi)$ are σ-formulas.

Formulas generated by rule i) are called *atoms* or *atomic formulas*, and the set of formulas generated by rules i)-v) is called *Predicate Language*. Just as in propositional logic, a *literal* or *elementary formula* is an atom or the negation of an atom and a *clause* is a disjunction of literals.

We say a variable occurring in a formula ϕ is *free* when it is not under the scope of any quantifier in ϕ, otherwise it shall be called *bound*.

Our goal now is to give a meaning to terms and formulas in the predicate language. In order to do so, the concept of interpretation is defined in the predicate logic.

Definition 2.15. (Semantics of predicate logic): Let σ be a vocabulary. A $\sigma - structure$ is a pair $A = (U_A, \mathcal{F}_A)$, where U_A is an arbitrary, non-empty set called the *Domain* or the *Universe* of A, and $\mathcal{F}_A$ is a mapping satisfying:

i. $\mathcal{F}_A$ maps every constant symbol c $\in \sigma$ to an element $\mathcal{F}_A$(c)=c^A of U_A.

ii. $\mathcal{F}_A$ maps every n-ary function symbol f in σ to an n-ary function $\mathcal{F}_A$(f)=f^A on U_A.

iii. $\mathcal{F}_A$ maps every n-ary predicate symbol P in σ to an n-ary relation $\mathcal{F}_A$(P)=P^A on U_A.

Definition 2.16. (Semantics of predicate logic): A $\sigma - interpretation$ I is a pair (A, π) where A is a σ-structure and $\pi : V \rightarrow U_I$, where V is the set of variables and U_I=U_A.

Definition 2.17. If I is an interpretation, x is a variable and $a \in U_I$, we define the interpretation $I_{[x/a]}$ as follows: $I_{[x/a]}$ coincides with I except in the variable x, where $I_{[x/a]}(x) = a$.

We now show how terms and formulas are evaluated under a certain interpretation I. We say an interpretation I is *suitable* for a set of formulas Φ if I is defined on all constant symbols, function symbols, predicate symbols and free variables occurring in formulas in Φ.

Constant symbols, function symbols and predicate symbols are evaluated by the use of $\mathcal{F}_A$. Bounded variables are interpreted by means of the quantifier affecting them and free variables are interpreted by means of π. The evaluation of a term will be an element of the domain, and the interpretation of a formula will be, as in propositional logic, a truth value.

Evaluation of terms: In order to evaluate a term t under an interpretation I, constant symbols, function symbols and free variables occurring in t are replaced by their corresponding interpretations. The result of the evaluation will be an element of the domain of I. We denote by I(t) or t^I the evaluation of the term t in I.

Evaluation of formulas: In order to evaluate a formula ϕ under an interpretation I, constant symbols and function symbols occurring in ϕ are replaced by their corresponding interpretations. A free occurrence of a variable x is replaced by its interpretation I(x)=$\pi(x)$, and bound occurrences of variables in ϕ are interpreted by means of the quantifiers affecting them, taking as a domain for the quantifiers the domain of the interpretation I. If the evaluation of ϕ is a true property we shall write I(ϕ)=T, if not we shall write I(ϕ)=F.

Now, in order to evaluate a formula ϕ in an interpretation I we proceed according to the following rules:

(1) If $\phi = Rt_1, \ldots, t_n$ then $I(\phi) = T$ iff $R^I I(t_1) \ldots I(t_n)$.
(2) $I(\neg\phi) = T$ iff $I(\phi) = F$
(3) $I(\phi \vee \psi) = T$ iff $I(\phi) = T$ or $I(\psi) = T$
(4) $I(\phi \wedge \psi) = T$ iff $I(\phi) = T$ and $I(\psi) = T$
(5) $I(\phi \rightarrow \psi) = T$ iff $I(\phi) = F$ or if $I(\phi) = I(\psi) = T$
(6) $I(\phi \leftrightarrow \psi) = T$ iff $I(\phi) = I(\psi)$
(7) $I(\exists x\phi) = T$ iff there is $a \in U_I$ such that $I_{[x/a]}(\phi) = T$
(8) $I(\forall x\phi) = T$ iff for every $a \in U_I$, $I_{[x/a]}(\phi) = T$

Definition 2.18. A formula ϕ is called *satisfiable* or *consistent* if there exists an interpretation I such that $I(\phi) = T$. In this case we say I is a *model* for ϕ.

A formula ϕ is called a *valid formula* or *tautology* if $I(\phi) = T$ for any interpretation I. If a formula ϕ is false under all interpretations it is said to be *inconsistent* or *unsatisfiable*.

Notice that ϕ is a valid formula if and only if $\neg\phi$ is inconsistent.

Definition 2.19. A set Φ of formulas is *satisfiable* if there exists an interpretation I such that $I(\phi) = T$ for every $\phi \in \Phi$.

Definition 2.20. Two σ-formulas ϕ and ψ are said to be *equivalent* if for every interpretation I $I(\phi) = I(\psi)$.

Definition 2.21. A formula ϕ is a *logical consequence* of formulas $\psi_1, \ldots, \psi_n$ if and only if $\psi_1 \wedge \ldots \wedge \psi_n \rightarrow \phi$ is a valid formula. It is easy to verify that this is equivalent to $\psi_1 \wedge \ldots \wedge \psi_n \wedge \neg\phi$ being unsatisfiable.

Definition 2.22. An interpretation I is said to be a *model* for a set of formulas Φ if I(ϕ)=T for every $\phi \in \Phi$.

2.3 Notation

i. $\Phi \overset{\circ}{\models} \psi$ (where Φ is a set of propositional formulas and ψ is a propositional formula) means ψ is a logical consequence of Φ.

ii. $I \overset{\circ}{\models} \phi$ means I is a model for ϕ in the propositional sense. $\overset{\circ}{\models} \phi$ means the propositional formula ϕ is a tautology.

iii. $\Phi \models \psi$ (where Φ is a set of formulas in the first order logic and ψ is a formula in the first order logic) means ψ is a logical consequence of Φ.

iv. $I \models \phi$ (with I being a first order interpretation and ϕ a first order formula) means I is a model for ϕ. $\models \phi$ means the first order formula ϕ is a tautology.

v. $\phi \overset{\circ}{\equiv} \psi$ means ϕ, ψ are logic-equivalent in Propositional logic.

vi. $\phi \equiv \psi$ means ϕ, ψ are logic-equivalent in Predicate logic.

vii. $Sat^{\circ}\phi$ means the propositional formula ϕ is satisfiable, in the same way $Sat^{\circ}\Phi$ means the set of propositional formulas Φ is satisfiable.

viii. $Sat\ \phi$ means the predicate formula ϕ is satisfiable, and $Sat\ \Phi$ means the set of predicate formulas Φ is satisfiable.

ix. $Uns^{\circ}\phi$ means the propositional formula ϕ is unsatisfiable, in the same way $Uns^{\circ}\Phi$ means the set of propositional formulas Φ is unsatisfiable.

x. $Uns\ \phi$ means the predicate formula ϕ is unsatisfiable, and $Uns\ \Phi$ means the set of predicate formulas Φ is unsatisfiable.

xi. Given a first order formula ϕ, voc(ϕ) is the set of constant symbols, function symbols and predicate symbols occurring in ϕ, free(ϕ) is the set of free variables occurring in ϕ, and nq(ϕ) is the number of quantifiers occurring in ϕ.

xii. Given an interpretation I and a first order formula ϕ, if $I \models \phi$ and the variables occurring in ϕ are among $v_1, \ldots, v_n$ and $I(v_i) = t_i$ with i=1,...,n, we shall write $I \models \phi[t_1, \ldots, t_n]$.

3 Herbrand's Theorem

Herbrand's theory is the first important approach to mechanical theorem proving, and was given in 1930 by Jaques Herbrand.

In the first order logic the difficulty arises from the fact that, since there are an infinite number of domains, in general there are an infinite number of interpretations of a formula. Now considering that by definition, a set Φ of formulas is unsatisfiable if and only if it is false under all interpretations over all domains, it is obvious that unlike in propositional logic it is not possible to verify the inconsistency of a set of formulas by evaluating it under all possible interpretations.
It would be convenient indeed to be able to do the same kind of verification considering only one (though infinite) domain. Fortunately such a domain exists, it is called the *Herbrand Universe of* Φ and will be introduced in this section.

3.1 Normal Forms

Formulas in the first order logic can be transformed into a normal form called the *prenex normal form*. The purpose of considering the prenex normal form of a formula is to simplify proof procedures, which will be discussed later.

Definition 3.1. A formula ϕ in the first order logic is said to be in *prenex form* if it has the form $\Theta_1 x_1 \dots \Theta_n x_n \psi$, where $\Theta_1, \dots, \Theta_n$ are quantifiers, and ψ is a formula containing no quantifiers. $\Theta_1 x_1 \dots \ \Theta_n x_n$ is called the *prefix* of ϕ and ψ the *matrix* of ϕ.
We call a formula *rectified* if no variable occurs both free and bounded, and if all quantifiers in the formula refer to different variables.

Theorem 3.2. *For every formula ϕ in the first order logic there exists a formula ϕ' in prenex normal form satisfying:*

i. $\phi \equiv \phi'$

ii. $voc(\phi) = voc(\phi')$

iii. $free(\phi) = free(\phi')$

iv. $nq(\phi) = nq(\phi')$.

Proof. By induction on the formula structure of ϕ)

- If ϕ is atomic, then it is in the desired form and $\phi' = \phi$.

- If $\phi = \neg\phi_1$, and $\psi_1 = \Theta_1 y_1 \dots \Theta_n y_n \psi_1'$ is the formula equivalent to ϕ_1 (existing by hypothesis) then $\phi \equiv \overline{\Theta_1} y_1 \dots \overline{\Theta_n} y_n \neg\psi_1'$ where $\overline{\Theta_i} = \exists$ if $\Theta_i = \forall$, and $\overline{\Theta_i} = \forall$ if $\Theta_i = \exists$. This formula has the desired form.

- If ϕ has the form $(\phi_1 \circ \phi_2)$ where $\circ \in \{\wedge, \vee\}$, then there are ψ_1, ψ_2 equivalent to ϕ_1, ϕ_2 in the desired form (by induction hypothesis). Variable sets in ψ_1 and ψ_2 can be made disjoint by renaming variables. If $\psi_1 = \Theta_1 x_1 \dots \Theta_n x_n \psi_1'$ and $\psi_2 = \Theta_1' x_1' \dots \Theta_l' x_l' \psi_2'$, then $\phi \equiv \Theta_1 x_1 \dots \Theta_n x_n \Theta_1' x_1' \dots \Theta_l' x_l' (\psi_1' \circ \psi_2')$, which is in the desired form.

- If ϕ has the form $\Theta x \phi_1$, where $\Theta \in \{\forall, \exists\}$, then ϕ_1 is equivalent (by induction hypothesis) to a formula of the form $\Theta_1 x_1 \dots \Theta_n x_n \phi_1'$. By renaming variables, we can suppose $x \neq x_i, i \in 1, \dots, n$ and $\phi \equiv \Theta x \Theta_1 x_1 \dots \Theta_n x_n \phi_1'$, which is in the desired form.

#

Remark 3.3. We can easily transform a given formula into its corresponding *prenex normal form* by applying the following algorithm:

1 Remove "$\leftrightarrow$" and "$\rightarrow$" by using the following equivalence rules:

$$\phi \leftrightarrow \psi \equiv (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$$
$$\phi \rightarrow \psi \equiv \neg\phi \vee \psi$$

.

2 Place "$\neg$" preceding the atomic formulas by using the following equivalence rules:

$$\neg\neg\phi \equiv \phi$$
$$\neg(\phi \vee \psi) \equiv \neg\phi \wedge \neg\psi$$
$$\neg(\phi \wedge \psi) \equiv \neg\phi \vee \neg\psi$$

3 Rename bound variables where necessary

4 Move the quantifiers to the left by using the following equivalence rules:

$$\neg\exists x \phi \equiv \forall x \neg\phi$$
$$\neg\forall x \phi \equiv \exists x \neg\phi$$

Example 3.4. Transform the formula $(\forall x)P(x) \to (\exists x)Q(x)$ into a prenex normal form.

$$(\forall x)P(x) \to (\exists x)Q(x) \equiv \neg((\forall x)P(x)) \vee (\exists x)Q(x) \equiv$$

$$\equiv (\exists x)(\neg P(x)) \vee (\exists x)Q(x) \equiv (\exists x)(\neg P(x) \vee Q(x))$$

Having discussed how to transform a given formula in the first order logic into a prenex normal form, we can now consider how to eliminate existential quantifiers. The purpose of doing so is to obtain a formula which is in an appropriate form for the algorithms presented in the next sections.

Definition 3.5. A formula ϕ is in *Skolem form* if ϕ is in *Prenex form* and the prefix of ϕ contains no existential quantifiers. ϕ is said to be in a *Skolem Standard form (SSF)* if it is in a Skolem form and its matrix is in CNF.

Before giving a theorem that ensures the possibility of transforming any given formula into a Skolem form, *substitutions* are briefly presented. A more general definition and description of their properties is left for the next section

Definition 3.6. Let ϕ be a formula, x a variable and t a term. Then $\phi[x/t]$ denotes the formula obtained from ϕ by replacing in ϕ every free occurrence of x by t.

Notation: By $I_{[x/u]}$ we denote an interpretation I', which is identical to I with the exception of the definition of $x^{I'}$: No matter whether I is defined on x or not, we let $x^{I'} = u$.

Theorem 3.7. *For every formula ϕ there exists a formula ϕ^* satisfying:*

1) ϕ^* is in Skolem form.

2) $voc(\phi) \subseteq voc(\phi^*)$ and $voc(\phi^*) \backslash voc(\phi)$ contains only function symbols and constant symbols.

3) Sat ϕ iff Sat ϕ^*

Proof. We can suppose ϕ is a rectified formula in prenex normal form. For every formula ϕ in prenex normal form, we can construct a formula ϕ^* satisfying 1) to 3) by induction over P(ϕ):=number of $\exists$ quantifiers in the prefix of ϕ:

- If P(ϕ)=0, then ϕ^*=ϕ and ϕ^* satisfies 1) to 3).
- Suppose now that for any formula ψ with P(ψ)=n we can construct a formula ψ^* satisfying 1) to 3). We must prove now that for any formula ϕ with $P(\phi) = n + 1$ there exists ϕ^* that satisfies 1) to 3).

  We have $\phi = \forall x_1 \ldots \forall x_m \exists y \Theta_1 z_1 \ldots \Theta_l z_l \chi$ where χ is the matrix of ϕ, and we consider m:=number of universal quantifiers preceding the first occurrence of an existential quantifier in the prefix of ϕ. Now:

  Let $t = f(x_1, ..., x_m)$, where f $\notin voc(\chi)$.

  Let $\phi' = \forall x_1 \ldots \forall x_m \Theta_1 z_1 \ldots \Theta_l z_l \chi[y/t]$. Now $P(\phi') = n$. We now establish $\phi^* = (\phi')^*$.

Clearly ϕ^* satisfies 1) and 2). To show that ϕ^* satisfies 3), since *Sat* ϕ' iff *Sat* $(\phi')^*$ by induction hypothesis, we prove that *Sat* ϕ iff *Sat* ϕ':

$\Leftarrow$) Let us suppose that ϕ' is satisfiable, so there is an interpretation I, suitable for ϕ', with I(ϕ')=T. Then, as voc(ϕ) $\subseteq voc(\phi')$, I is also suitable for ϕ so:

for all $u_1, \ldots, u_m \in U_I$,

$$I_{[x_1/u_1]\ldots[x_m/u_m]}(\Theta_1 z_1 \ldots \Theta_l z_l \chi[y/f(x_1, \ldots, x_m)]) = T$$

hence for all $u_1, \ldots, u_m \in U_I$,

$$I_{[x_1/u_1]\ldots[x_m/u_m][y/v]}(\Theta_1 z_1 \ldots \Theta_l z_l \chi) = T$$

where v=f($u_1, \ldots, u_m$). Therefore we get that for all $u_1, \ldots, u_m \in U_I$ there exists v$\in U_I$ such that

$$I_{[x_1/u_1]\ldots[x_m/u_m][y/v]}(\Theta_1 z_1 \ldots \Theta_l z_l \chi) = T$$

hence

$$I(\forall x_1 \ldots \forall x_m \exists y \Theta_1 z_1 \ldots \Theta_l z_l \chi) = T$$

and thus $I \models \phi$, so ϕ is satisfiable.

$\Rightarrow$) Conversely, suppose ϕ has a model I. We can assume that I is undefined on function symbols that do not occur in ϕ. Hence I is not defined f and not suitable for ϕ'. Since $I(\phi) = T$, we have:

for all $u_1, \ldots, u_m \in U_I$ there exists $v \in U_I$ such that

$$(*) \quad I_{[x_1/u_1]\ldots[x_m/u_m][y/v]}(\Theta_1 z_1 \ldots \Theta_l z_l \chi) = T$$

Now we define a new interpretation I', identical to I except for the fact I' is defined on f. Let $f^{I'}$ be defined as follows:

$$f^{I'}(u_1, \ldots, u_m) = v$$

where v is chosen according to (*). Thus we obtain that for all $u_1, \ldots, u_m \in U_I$

$$I'_{[x_1/u_1]\ldots[x_m/u_m][y/f^{I'}(u_1,\ldots,u_m)]}(\Theta_1 z_1 \ldots \Theta_l z_l \chi) = T$$

therefore, for all $u_1, \ldots, u_m \in U_I$

$$I'_{[x_1/u_1]\ldots[x_m/u_m]}(\Theta_1 z_1 \ldots \Theta_l z_l \chi[y/f(x_1, \ldots, x_m)]) = T$$

so

$$I'(\forall x_1 \ldots \forall x_m \Theta_1 z_1 \ldots \Theta_l z_l \chi[y/f(x_1, \ldots, x_m)]) = T$$

hence I'$\models \phi'$ so ϕ' is satisfiable.

#

Remark 3.8. Notice that the transformation of a formula to Skolem form does not preserve equivalence (because of the new function symbols occurring).

Remark 3.9. Since the matrix of a formula in Skolem form contains no quantifiers, it can be transformed into a CNF form.

Given a formula ϕ in predicate logic (with possible occurrences of free variables), the following algorithm will produce a formula ϕ^* in SSF satisfying Sat ϕ iff Sat ϕ^*:

1. Rectify ϕ by systematic renaming of bound variables, this is, rename variables so that no variable occurs both free and bound. The result is a formula $\phi_1 \equiv \phi$.
2. Produce from ϕ_1 an equivalent formula ϕ_2 in prenex form.
3. Delete existential quantifiers in ϕ_2 by transforming it into a Skolem formula ϕ_3. So we will have Sat ϕ_3 iff Sat ϕ_2.
4. Convert the matrix of ϕ_3 into CNF.

Example 3.10. Obtain a standard form for the formula

$(\forall x)(\exists y)(\exists z)((\neg P(x, y) \wedge Q(x, z)) \vee R(x, y, z))$

First the matrix is transformed into a CNF:

$(\forall x)(\exists y)(\exists z)((\neg P(x, y) \vee R(x, y, z)) \wedge (Q(x, y) \vee R(x, y, z)))$

Then, since $\exists y$ and $\exists z$ are both preceded by $\forall x$, the existential variables y and z are replaced, respectively, by one-place functions f(x) and g(x). Thus we obtain the following standard form of the formula:

$(\forall x)((\neg P(x, f(x)) \vee R(x, f(x), g(x))) \wedge (Q(x, g(x)) \vee R(x, f(x), g(x))))$

Notation 3.11. If the variables occurring in a formula ϕ are among $x_1, \ldots, x_n$ we shall write $\phi(x_1, \ldots, x_n)$.

If $\Phi = \{\phi_1(x_1, \ldots, x_n), \ldots, \phi_k(x_1, \ldots, x_n)\}$ is a finite set of clauses, we shall write $\alpha_\Phi = \forall x_1 \ldots \forall x_n(\phi_1 \wedge \ldots \wedge \phi_k)$. This way we have that the set $\{\alpha_\Phi : \Phi$ is a finite set of clauses$\}$ coincides with the set of Skolem Standard Forms.

3.2 Herbrand's Theorem

In the proofs of Herbrand's Theory, we shall use propositional interpretations along with first order interpretations when we consider sets of formulas in the predicate logic. First the following definition must be given:

Definition 3.12. Let Σ be a set of predicate logic formulas. We define the *Boolean closure of* Σ, in symbols $< \Sigma >$, as the set of formulas generated by applying the following rules:

(R1) For every $\phi \in \Sigma$, then $\phi \in < \Sigma >$

(R2) If $\phi \in < \Sigma >$, then $(\neg\phi) \in < \Sigma >$

(R3) If $\phi, \psi \in < \Sigma >$, then $(\phi \vee \psi) \in < \Sigma >$

(R4) If $\phi, \psi \in < \Sigma >$, then $(\phi \wedge \psi) \in < \Sigma >$

(R5) If $\phi, \psi \in < \Sigma >$, then $(\phi \rightarrow \psi) \in < \Sigma >$

(R6) If $\phi, \psi \in < \Sigma >$, then $(\phi \leftrightarrow \psi) \in < \Sigma >$

We can now define propositional interpretations for $< \Sigma >$, taking Σ as a set of proposition symbols.

Definition 3.13. A propositional interpretation for Σ is a mapping $I : \Sigma \rightarrow \{T, F\}$.

Example 3.14. Let $\Sigma = \{\exists x P(x), \forall x P(x), R(x, y)\}$. Then, the assignment

$I(\exists x P(x)) = F$

$I(\forall x P(x)) = T$

$I(R(x, y)) = T$

is a propositional interpretation for $< \Sigma >$.

Proposition 3.15. *Let Σ be a set of formulas in the predicate logic. Let $\phi \in < \Sigma >$. Then:*

(a) If Sat ϕ, then Sat$^\circ\phi$

(b) If $\overset{\circ}{\models} \phi$, then $\models \phi$

Proof. (a) For every interpretation I in the first order logic, we define the propositional interpretation I^* as follows:

if $\phi \in \Sigma$, $I^*(\phi) = T$ if $I \models \phi$, and $I^*(\phi) = F$ if $I \not\models \phi$.

Now if we have $Sat\ \phi$, then there exists I such that $I \models \phi$, hence $I^* \models \phi$ and so $Sat^\circ \phi$.

(b) $\overset{\circ}{\models} \phi$, hence $Uns^\circ\ \neg\phi$ and by (a) $Uns\ \neg\phi$, therefore $\models \phi$. #

Having defined the previous concepts, Herbrand's theory can now be introduced.

Definition 3.16. Let Φ be a finite set of clauses. Let $\tau = voc(\Phi)$. We define:

$H_0 = \{c \in \tau : \text{c is a constant symbol}\}$ if τ has any constant symbols, $H_0 = \{c\}$ where c is a new constant symbol if not.

$H_{i+1} = H_i \cup \{f(t_1, \ldots, t_n) : t_1, \ldots, t_n \in H_i,\ \text{f is an n-arity function symbol in } \tau\}$

Let $H^* = \bigcup_{i \geq 0} H_i$. We call H^* the *Herbrand Universe* of Φ.

Notation: We put $H^* = H^*(\Phi)$.

Example 3.17. Let $\Phi = \{P(y), \neg P(x) \vee P(f(x))\}$. Then

$H_0 = \{a\}$

$H_1 = \{a, f(a)\}$

$H_2 = \{a, f(a), f(f(a))\}$

$\vdots$

$H^* = \{a, f(a), f(f(a)), f(f(f(a))), \ldots\}$

Example 3.18. Let $\Phi = \{P(f(x), a, g(y), b)\}$

$H_0 = \{a, b\}$

$H_1 = \{a, b, f(a), f(b), g(a), g(b)\}$

$H_2 = \{a, b, f(a), f(b), g(a), g(b), f(f(a)), f(f(b)), f(g(a)), f(g(b)), g(f(a)), g(f(b)), g(g(a)), g(g(b))\}$

$\vdots$

Notation 3.19. We will write $voc^*(\Phi) = voc(\Phi) \cup H_0$.

Definition 3.20. Let Φ be a finite set of clauses. Let $H^* = H^*(\Phi)$. Let $\phi \in \Phi$. A *ground instance of* ϕ is a formula obtained by replacing the variables in ϕ by members of H^*.

Definition 3.21. Let Φ be a finite set of clauses. Let $H^* = H^*(\Phi)$. Let $\tau = voc^*(\Phi)$. We define the *Herbrand Base* of Φ as:

$B = \{P(t_1, \ldots, t_n) : \text{P is an n-arity predicate symbol in } \tau, t_1, \ldots, t_n \in H^*\}$. We shall call the formulas in B and their negations *B-elementary formulas.*

Example 3.22. Let $\Phi = \{P(x), Q(f(y)) \vee R(y)\}$. Then $H^* = \{a, f(a), f(f(a)), \ldots\}$ is the Herbrand Universe of Φ. C=P(x) is a clause in Φ and P(a) and P(f(f(a))) are both ground instances of C.

In the sequel, for a finite set of clauses Φ, $H^*(\Phi)$ will be used as the universe to search for potential models for Φ, and it will be shown that this results in no loss in generality.

Definition 3.23. Let Φ be a finite set of clauses. Let $\tau = voc^*(\Phi)$. Let $H^* = H^*(\Phi)$. We say a τ^*-interpretation H is a *Herbrand interpretation* or *H-interpretation* if:

i. The domain of H is H^*.

ii. $c^H = c$ for every constant symbol $c \in \tau$

iii. For every n-arity function symbol f in τ, $f^H : (H^*)^n \longrightarrow H^*$ is defined by: $f^H(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$ for all $t_1, \ldots, t_n \in H^*$.

Remark 3.24. For Herbrand interpretations, the domain and the assignments for constant symbols and function symbols are restricted, though the assignments for predicate symbols can be chosen freely. Thus it is natural to consider the following definition.

Definition 3.25. Let Φ be a finite set of clauses. Let B be the Herbrand base of Φ. A *B-interpretation* is a mapping $I : B \longrightarrow \{V, F\}$.

A B-interpretation I can be represented as: $\{\phi \in B : I(\phi) = T\} \cup \{\neg\phi : \phi \in B, I(\phi) = F\}$.

Remark 3.26. Every Herbrand's interpretation H is determined by the truth value of the formulas of B in H . Therefore there is a one-to-one correspondence between B-interpretations and H-interpretations.

Definition 3.27. Given a finite set of clauses Φ, with $\tau = voc^*(\Phi)$, we shall assign a *Herbrand Interpretation* H_I to every τ-Interpretation I in the following way:

$$R^{H_I}(t_1, \ldots, t_n) \text{ iff } R^I(t_1^I, \ldots, t_n^I)$$

for every n-arity predicate symbol R in τ and every $t_1, \ldots, t_n \in H^*$

Proposition 3.28. *Let Φ be a finite set of clauses. Let $\tau = voc^*(\Phi)$. For every τ-interpretation I, let H_I be the Herbrand Interpretation corresponding to I. Then: $I \models \alpha_\Phi$ implies $H_I \models \alpha_\Phi$.*

Proof. Let $H^* = H^*(\Phi)$. Let I be a τ-interpretation such that $I \models \alpha_\Phi$. Let l be a natural number such that for every $\phi \in \Phi$ the variables in ϕ are among $v_1, \ldots, v_l$. We prove the following assertion:

(*) For every $\phi \in \Phi$ and for every $t_1, \ldots, t_l \in H^*$, $H_I \models \phi[t_1, \ldots, t_l]$

from which we conclude that $H_I \models \alpha_\Phi$.

Lets prove (*). Let $\phi = \psi_1 \vee \ldots \vee \psi_m \in \Phi$. Since $I \models \alpha_\Phi$,
$I \models \phi[t_1^I, \ldots, t_l^I]$ for all $t_1, \ldots, t_l \in H^*$, therefore
$I \models \psi_i[t_1^I, \ldots, t_l^I]$ for some $i \in \{1, \ldots, m\}$,
and since ψ_i is elementary $H_I \models \psi_i[t_1, \ldots, t_l]$, so $H_I \models \phi[t_1, \ldots, t_l]$. #

Example 3.29. Consider a finite set of clauses $\Phi = \{P(x), Q(y, f(y, a))\}$ and the interpretation I, suitable for Φ, defined as follows:

-The domain is D=$\{1, 2\}$.

-The assignments for constant symbols and function symbols are: a=2, f(1,1)=1, f(1,2)=2, f(2,1)=2, f(2,2)=1

-The assignments for predicate symbols are: P(1)=T, P(2)=F, Q(1,1)=F, Q(1,2)=T, Q(2,1)=F, Q(2,2)=T

The Herbrand Interpretation H_I corresponding to the interpretation I is defined as follows:

First we construct the Herbrand Base of Φ:

B=$\{P(a), Q(a, a), P(f(a, a)), Q(a, f(a, a)), Q(f(a, a), a), Q(f(a, a), f(a, a)), \ldots\}$

Next we evaluate each member of B by using the assignments:

P(a)=P(2)=F

Q(a,a)=Q(2,2)=T

P(f(a,a))=P(f(2,2))=P(1)=T

Q(a,f(a,a))=Q(2,f(2,2))=Q(2,1)=F

Q(f(a,a),a)=Q(f(2,2),2)=Q(1,2)=T

Q(f(a,a),f(a,a))=Q(f(2,2),f(2,2))=Q(1,1)=F

$\vdots$

Therefore, the H-interpretation corresponding to I is

$H_I = \{\neg P(a), Q(a,a), P(f(a,a)), \neg Q(a, f(a,a)),$
$Q(f(a,a),a), \neg Q(f(a,a), f(a,a)), \ldots\}$

Theorem 3.30. *Let Φ be a finite set of clauses. Then*

$Uns\ \alpha_\Phi$ iff $H \models \neg\alpha_\Phi$ for every Herbrand Interpretation H

Proof. $\Rightarrow$) Obviously, if α_Φ is unsatisfiable, every Herbrand Interpretation H will be a model for $\neg\alpha_\Phi$.

$\Leftarrow$) Let us suppose that for every Herbrand Interpretation H, H$\models \neg\alpha_\Phi$. If α_Φ was to have a model I, let H_I be the Herbrand interpretation associated with I according with Definition 3.27. Then by Proposition 3.28 $H_I \models \alpha_\Phi$ and we would have a contradiction.

#

Notation 3.31. We denote by Φ' the set of ground instances of clauses of Φ.

Theorem 3.32. *(Herbrand's Theorem):*

$Uns\ \alpha_\Phi$ iff there exists a finite subset $\Phi'_0 \subseteq \Phi'$ such that $Uns^\circ\Phi'_0$.

Proof. By using Proposition 3.28, the relationship between H-interpretations and B-interpretations and the Compactness Theorem for propositional logic we have:
$Uns\ \alpha_\Phi \iff$
for every Herbrand interpretation H, $H \models \neg\alpha_\Phi \iff$
for every B-interpretation I, $H_I \models \neg\alpha_\Phi \iff$
for every B-interpretation I there exist $t_1, \ldots, t_n \in H^*$ and $\phi \in \Phi$ such that
$H_I \models \neg\phi[t_1, \ldots, t_n] \iff$
for every B-interpretation I there exist $t_1, \ldots, t_n \in H^*$ and $\phi \in \Phi$ such that

$H_I \overset{\circ}{\models} \neg\phi_{[v_1/t_1]\ldots[v_n/t_n]} \Longleftrightarrow$
for every B-interpretation I there exists $\phi' \in \Phi'$ such that $I \overset{\circ}{\not\models} \phi' \Longleftrightarrow$
$Uns^{\circ}\Phi' \Longleftrightarrow$
there exists a finite $\Phi'_0 \subseteq \Phi'$ such that $Uns^{\circ}\Phi'_0$.

#

So Herbrand's Theorem asserts that the unsatisfiability of a formula in the first order logic can be proved by showing the unsatisfiability of finite sets of propositional formulas.

Example 3.33. Let the set Φ consist of the following clauses:

$\Phi = \{\neg P(x,y,u) \vee \neg P(y,z,v) \vee \neg P(x,v,w) \vee P(u,z,w),$
$\neg P(x,y,u) \vee \neg P(y,z,v) \vee \neg P(u,z,w) \vee P(x,v,w),$
$P(g(x,y),x,y),\ P(x,h(x,y),y),\ P(x,y,f(x,y)), \neg P(k(x),x,k(x))\}$

This set is unsatisfiable. However it is not easy to find by hand a finite unsatisfiable set Φ'_0 of ground instances of clauses in Φ. The following is a desired set Φ'_0:

$\Phi'_0 = \{P(a,h(a,a),a),\ \neg P(k(h(a,a)),h(a,a),k(h(a,a))),$
$P(g(a,k(h(a,a))),a,k(h(a,a))),$
$\neg P(g(a,k(h(a,a))),a,k(h(a,a))) \vee \neg P(a,h(a,a),a)$
$\wedge \neg P(g(a,k(h(a,a))) \vee P(k(h(a,a)),h(a,a),k(h(a,a)))\}$

3.3 Gilmore's Implementation

Let Φ be a finite set of clauses. Let $H^* = H^*(\Phi)$, and $P \subseteq H^*$. Let B be the Herbrand base of Φ.

Notation 3.34. We denote by $P(\Phi)$ the set of instances of formulas in Φ with terms in P.

Now we have the following corollary of Herbrand's Theorem:

Corollary 3.35. *$Uns\ \alpha_\Phi$ iff there exists a finite set $P \subseteq H^*$ such that $Uns^{\circ}\ P(\Phi)$.*

In Gilmore's implementation we successively generate the sets $H_0 \subseteq H_1 \subseteq \ldots$ such that $H^* = \bigcup_{i \geq 0} H_i$, and then test $H_i(\Phi)$ for unsatisfiability by using the multiplication method. This consists in writing $H_i(\Phi)$ out in a disjunctive normal form, and deleting every conjunction in it containing a complementary pair. Should an empty set be obtained, then $H_i(\Phi)$ is unsatisfiable and a proof is found.

The multiplication method used by Gilmore is inefficient. It is enough to consider the following finite set of clauses:

$\Phi = \{P(x, g(x), y, h(x, y), z, k(x, y, z)), \neg P(u, v, l(v), w, f(u, w), x)\}$.

Φ is unsatisfiable, though the least i such that $H_i(\Phi)$ is unsatisfiable is 5 and $|H_5| \approx 10^{64}$ and $|H_5(\Phi)| \approx 10^{256}$.

For this reason it would be inefficient to use Herbrand's Theorem to prove the unsatisfiability of a formula, and other procedures should be considered. Resolution is one of them and is introduced in the next chapter.

4 Robinson's Theorem

Resolution is a syntactic transformation applied to formulas. Providing resolution is applicable to two given formulas, a third formula is generated and can be used in further resolution steps, giving place to the *Resolution Calculus*.

Our aim in this section is to prove that the Resolution Calculus is correct and complete.

4.1 Resolution for Propositional Logic

Now Resolution for Propositional Logic will be presented. Before introducing the concept of resolvent some definitions must be given.

Definition 4.1. Given a clause ϕ, we define $Mb(\phi)$ as the set of literals that occur in ϕ.

Definition 4.2. Given a literal ϕ, we define $\sim \phi = \neg\phi$ if ϕ is an atom, and $\sim \phi = \psi$ if $\phi = \neg\psi$.

Definition 4.3. Suppose that ϕ_1, ϕ_2, ϕ are clauses. We say that ϕ is a *resolvent* of ϕ_1 and ϕ_2 if there exists $\psi \in Mb(\phi_1)$ such that $\sim \psi \in Mb(\phi_2)$ and

$$Mb(\phi) = (Mb(\phi_1)\backslash\{\psi\}) \bigcup (Mb(\phi_2)\backslash\{\sim \psi\})$$

.

Example 4.4. If $\phi_1 = \neg P \vee Q \vee R$ and $\phi_2 = \neg Q \vee S$, we have that $\phi = \neg P \vee R \vee S$ is a resolvent of ϕ_1, ϕ_2.

Notation 4.5. If σ is a vocabulary and I is a σ-interpretation, we denote I by

$\{P \in \sigma : I(P) = T\} \cup \{\neg P : P \in \sigma, I(P) = F\}$

An important property is that a resolvent of two clauses is a logical consequence of the clauses:

Proposition 4.6. *If ϕ is a resolvent of ϕ_1 and ϕ_2 then $\{\phi_1, \phi_2\} \models \phi$.*

Proof. $\phi_1 \overset{\circ}{\equiv} \phi_1' \vee \psi$, $\phi_2 \overset{\circ}{\equiv} \phi_2' \vee \neg\psi$, $\phi \overset{\circ}{\equiv} \phi_1' \vee \phi_2'$. Let I be an interpretation such that $I \overset{\circ}{\models} \phi_1 \wedge \phi_2$.

Case 1: $\psi \in I$. Since $I \overset{\circ}{\models} \phi_2'$ we have $I \overset{\circ}{\models} \phi$.

Case 2: $\neg\psi \in I$. Since $I \overset{\circ}{\models} \phi_1'$ then $I \overset{\circ}{\models} \phi$. #

Definition 4.7. If Φ is a finite set of clauses we define

$$R(\Phi) = \Phi \cup \{\phi : \text{there are } \phi_1, \phi_2 \in \Phi \text{ s.t. } \phi \text{ is a resolvent of } \{\phi_1, \phi_2\}\}$$

· Thus we define also $R^0 = \Phi$, $R^{n+1} = R(R^n(\Phi))$ for every n≥0, and $R^*(\Phi) = \bigcup_{i \geq 0} R^i(\phi)$.

Definition 4.8. We denote by $\square$ the *empty clause* or *empty disjunction*. The truth evaluation of the empty clause is always false, as a clause corresponds to a disjunction of literals and thus is evaluated as true only when at least one of them is true.

Example 4.9. Let $\Phi = \{\neg P \vee Q, \neg Q, P\}$. Then, we have:

$R^0(\Phi) = \Phi$

$R^1(\Phi) = \{\neg P \vee Q, \neg Q, P\} \cup \{\neg P, Q\}$

$R^2(\Phi) = \{\neg P \vee Q, \neg Q, P, \neg P, Q, \square\}$

Remarks 4.10.

1 $\Phi \overset{\circ}{\models} R(\Phi) \overset{\circ}{\models} R^2(\Phi) \ldots$

2 As ϕ is finite, there exists an n_0 such that for every $n \geq n_0$, then $R^n(\Phi) = R^{n_0}(\phi)$.

Definition 4.11.

(a) A proof by resolution of ϕ from Φ is a finite sequence $(\phi_1, \ldots, \phi_l)$ s.t. $\phi_l = \phi$ and for all $i \in \{1, \ldots, l\}, \phi_i \in \Phi$ or there exist $j, k \in \{1, \ldots, i-1\}$ s.t. ϕ_i is a resolvent of ϕ_j, ϕ_k.

(b) A refutation of Φ by resolution is a proof by resolution of $\Box$ from Φ.

Notation 4.12. We will write $\Phi \underset{R}{\vdash} \phi$ iff there is a proof by resolution of ϕ from Φ.

Proposition 4.13. $\Phi \underset{R}{\vdash} \phi$ *iff* $\phi \in R^n(\Phi)$ *for some* $n \geq 0$.

Proof. $\Rightarrow$) Easily proved by induction on the length of a proof by resolution.

$\Leftarrow$) We proceed by induction on n.

If n=0, $\phi \in R^0(\Phi) = \Phi$, so obviously $\Phi \underset{R}{\vdash} \phi$.

Now supposing the assertion stands for n, we prove n+1: $\phi \in R^{n+1}(\Phi)$, therefore $\phi \in R^n(\Phi)$ (in which case by the I.H. $\Phi \underset{R}{\vdash} \phi$), or ϕ is a resolvent of two clauses $\phi_1, \phi_2 \in R^n(\Phi)$, so by the induction hypothesis $\Phi \underset{R}{\vdash} \phi_1, \Phi \underset{R}{\vdash} \phi_2$ therefore $\Phi \underset{R}{\vdash} \phi$.
#

Examples 4.14. (1) $\{\neg P \vee Q, \neg Q, P\} \underset{R}{\vdash} \Box$

(2)$\{P \vee Q, \neg P \vee Q, P \vee \neg Q, \neg P \vee \neg Q\} \underset{R}{\vdash} \Box$

In the following theorem we proceed to the proof of correctness and completeness of the resolution calculus (with respect to unsatisfiability).

Theorem 4.15. ***Basic Resolution Theorem:*** *If* Φ *is a finite set of clauses, then*

$$Uns^{\circ}\Phi \text{ iff } \Box \in R^n(\Phi) \text{ for some } n \geq 0.$$

Proof. $\Leftarrow$) $\Box \in R^n(\Phi)$ therefore $\Phi \underset{R}{\vdash} \Box$, hence $Uns^{\circ}\Phi$ as $\Box$ is unsatisfiable.

$\Rightarrow$) Suppose Φ is unsatisfiable. We show that $\Box \in R^*(\Phi)$ by induction on the number n of different atomic formulas in the clauses of Φ.

If n=0, then it must be that $\Phi = \{\Box\}$, and therefore $\Box \in R^*(\Phi)$.

Now let n$\geq$1. Suppose that for every unsatisfiable set of clauses Ψ containing at most the atomic formulas $\phi_1, \ldots, \phi_n$, we have that $\Box \in R^*(\Psi)$. Now let Φ

be a clause set containing the atomic formulas $\phi_1, \ldots, \phi_n, \phi_{n+1}$. Without loss of generality we may assume that no clause contains both ϕ_{n+1} and $\neg\phi_{n+1}$. From Φ we obtain two new set of clauses Φ_1 and Φ_2 as follows:

Φ_1 results from Φ by canceling every occurrence of the positive literal ϕ_{n+1} within a clause, and for every occurrence of the negative literal $\neg\phi_{n+1}$ the entire clause is canceled. Analogously Φ_2 is defined where the roles of ϕ_{n+1} and $\neg\phi_{n+1}$ are interchanged.

Notice that Φ_1 (Φ_2) essentially results from Φ by fixing the assignment of ϕ_{n+1} to F (T). Therefore both Φ_1 and Φ_2 are unsatisfiable. To prove this fact, assume to the contrary that Φ_1 has a satisfying interpretation $I = \{\phi_1, \ldots, \phi_n\} \longrightarrow \{T, F\}$. Then we could find a satisfying interpretation I' for Φ, where $I'(\chi) = I(\chi)$ if $\chi \in \{\phi_1, \ldots, \phi_n\}$, and $I'(\chi) = F$ if $\chi = \phi_{n+1}$. This contradicts the unsatisfiability of Φ. Similarly it can be shown that ϕ_2 is unsatisfiable.

Therefore, by induction hypothesis, $\Box \in R^*(\Phi_1)$ and $\Box \in R^*(\Phi_2)$. This means that there is a sequence of clauses $\psi_1, \ldots, \psi_m$ such that:

$\psi_m = \Box$, and for $i = 1, \ldots, m$ we have $\psi_i \in \Phi_1$ or ψ_i is a resolvent of two clauses ψ_a, ψ_b with $a, b < i$.

An analogous sequence $\psi'_1, \ldots, \psi'_l$ exists for Φ_2.
Some of the clauses ψ_i were obtained from Φ by canceling the literal ϕ_{n+1}. By restoring the original clauses $\psi_i \vee \phi_{n+1}$, and carrying ϕ_{n+1} along in the resolution steps, we obtain from $\psi_1, \ldots, \psi_m$ a new proof sequence for Φ which witnesses that $\phi_{n+1} \in R^*(\Phi)$. Similarly, restoring $\neg\phi_{n+1}$ in the sequence $\psi'_1, \ldots, \psi'_l$ shows that $\neg\phi_{n+1} \in R^*(\Phi)$. Now by a further resolution step on clauses $\phi_{n+1}, \neg\phi_{n+1}$ the empty clause can be derived, and therefore $\Box \in R^*(\Phi)$. #

Corollary 4.16. *Let $\phi_1, \ldots, \phi_n, \phi$ be propositional formulas. Let $\psi_1 \wedge \ldots \wedge \psi_k$ be a CNF of $\phi_1 \wedge \ldots \wedge \phi_n \wedge \neg\phi$. Then*

$$\{\phi_1, \ldots, \phi_n\} \overset{\circ}{\models} \phi \text{ iff } \{\psi_1, \ldots, \psi_k\} \underset{R}{\vdash} \Box$$

.

Proof. $\{\phi_1, \ldots, \phi_n\} \overset{\circ}{\models} \phi$, or equivalently $Unsat^\circ \phi_1 \wedge \ldots \wedge \phi_n \wedge \neg\phi$, by the Basic Resolution Theorem this is equivalent to $\Box \in R^*(\{\psi_1, \ldots, \psi_k\})$ which by proposition 4.13. is equivalent to $\{\psi_1, \ldots, \psi_k\} \underset{R}{\vdash} \Box$. #

4.2 Resolution for Predicate Logic

The aim of this section is to develop the Resolution Method for Predicate Logic. First, we want to define the notion of resolvent of two clauses in the first order logic. For this, some previous definitions are needed.

Definition 4.17. A *substitution* is a set of the form $\{x_1/t_1, \ldots, x_n/t_n\}$ where $x_1, \ldots, x_n$ are pairwise different variables and $t_1, \ldots, t_n$ are terms. We denote as ϵ the empty substitution.

We shall call the elementary formulas and terms *expressions*.

Let ω be an expression and $\lambda = \{x_1/t_1, \ldots, x_n/t_n\}$ a substitution. We denote as $\omega\lambda$ the expression obtained by replacing every occurrence of x_i by t_i (i=1,...,n).

Definition 4.18. *Composition of substitutions:* Let $\lambda = \{x_1/t_1, \ldots, x_n/t_n\}$ and $\sigma = \{y_1/s_1, \ldots, y_m/s_m\}$ be two substitutions. We define $\lambda \circ \sigma$ as $\lambda' \cup \sigma'$ where:

$$\lambda' = \{x_i/t_i\sigma : i \in \{1, \ldots, n\}, t_i\sigma \neq x_i\}$$

$$\sigma' = \{y_j/s_j : j \in \{1, \ldots, m\}, y_i \notin \{x_1, \ldots, x_n\}\}$$

.

Example 4.19. Let $\theta = \{y/f(x), z/c\}, \lambda = \{x/g(y), z/d\}$. Then $\theta \circ \lambda = \{y/f(g(y)), z/c, x/g(y)\}$

Definition 4.20. Let τ be a vocabulary. We denote by $(A_\tau)^*$ the set of expressions with respect to the vocabulary τ.

Proposition 4.21. *Let θ, ζ, γ be substitutions, and ω be an expression. Then:*

1 $\theta \circ \epsilon = \epsilon \circ \theta = \theta$.

2 $(\omega\theta)\zeta = (\omega)\theta\zeta$ *for every* $\omega \in (A_\tau)^*$.

3 If $\omega\theta = \omega\zeta$ *for every* $\omega \in (A_\tau)^*$, *then* $\theta = \zeta$.

4 $(\theta\zeta)\gamma = \theta(\zeta\gamma)$.

Proof. 1) Immediate.

2) Let $\theta = \{x_1/t_1, \ldots, x_n/t_n, y_1/s_1, \ldots, y_k/s_k\}$,
$\zeta = \{y_1/r_1, \ldots, y_k/r_k, z_1/q_1, \ldots, z_m/q_m\}$, where no pair of the variables $x_1, \ldots, x_n, y_1, \ldots, y_k, z_1, \ldots, z_m$ are equal. Let $\omega(x_1, \ldots, x_n, y_1, \ldots, y_k, z_1, \ldots, z_m)$ be an expression in $(A_\tau)^*$. Then $\omega\theta = \omega(t_1, \ldots, t_n, s_1, \ldots, s_k, z_1, \ldots, z_m)$, and $(\omega\theta)\zeta = \omega(t_1\zeta, \ldots, t_n\zeta, s_1\zeta, \ldots, s_k\zeta, z_1\zeta, \ldots, z_m\zeta)=$
$\omega(t_1\zeta, \ldots, t_n\zeta, s_1\zeta, \ldots, s_k\zeta, q_1, \ldots, q_m) = (\omega)\theta\zeta$, by the definition of $\theta \circ \zeta = \{x_1/t_1\zeta, \ldots, x_n/t_n\zeta, y_1/s_1\zeta, \ldots, y_k/s_k\zeta, z_1/q_1, \ldots, z_m/q_m\}$.

3) Let $z_1, \ldots, z_m$ be the variables in θ and ζ. $\omega\theta = \omega\zeta$ for every expression ω implies that $z_i\theta = z_i\zeta$ for every $i \in \{1, \ldots, m\}$ so $\theta = \zeta$.

4) For every $\omega \in (A_\tau)^*$: $\omega((\theta \circ \zeta) \circ \gamma) = \omega((\theta \circ \zeta))\gamma = ((\omega\theta)\zeta)\gamma = (\omega\theta)(\zeta \circ \gamma) = \omega(\theta \circ (\zeta \circ \gamma))$, where each equality is due to (2). Now (3) applies.

#

Notation 4.22. If $\Omega \subseteq (A_\tau)^*$ and λ is a substitution, we put $\Omega\lambda = \{\omega\lambda : \omega \in \Omega\}$.

Definition 4.23. Let Ω be a finite and non empty set of expressions. Let θ be a substitution. We say θ is a *unifier* for Ω if $|\Omega\theta| = 1$.

Definition 4.24. Let Ω be a finite non empty set of expressions. Let σ be a unifier for Ω. We say σ is a *most general unifier* if for every unifier θ for Ω there exists a substitution λ such that $\theta = \sigma \circ \lambda$.

Definition 4.25. A finite non empty set of expressions Ω is *unifiable* if there exists a unifier for Ω.

Definition 4.26. Let Ω be a finite non empty set of expressions. We define the *disagreement set of* Ω, in symbols $\mathrm{D}(\Omega)$, as the set of all sub-expressions starting at the first position where the expressions in Ω differ. When no confusion is possible, we will write D=D(Ω).

Example 4.27. Let $\Omega = \{P(f(f(x)), g(x, y)), P(f(g(c, d)), g(c, x)\}$. Then the disagreement set for Ω is $D = \{f(x), g(c, d)\}$.

Having set these concepts, the goal will be to set an algorithm to find most general unifiers for unifiable sets of expressions, and use the algorithm to prove that every unifiable set of expressions has a most general unifier.

Unification Algorithm

The input is a non empty set of expressions Ω. The algorithm produces sets $\Omega_0, \ldots, \Omega_k, \ldots$ where $\Omega_0 = \Omega$, and substitutions $\sigma_0, \ldots, \sigma_k, \ldots$ where $\sigma_0 = \epsilon$.

STEP 1. Set k:=0, $\Omega_k := \Omega$, $\sigma_k := \epsilon$

STEP 2. If $|\Omega_k| = 1$, then $\sigma := \sigma_k$ and STOP 1.

STEP 3. If $|\Omega_k| > 1$ find the disagreement set D_k of Ω_k.

-If there do not exist $x_k, t_k \in D_k$ such that x_k does not occur in t_k then STOP 2.

-If there exist $x_k, t_k \in D_k$ such that x_k does not occur in t_k then: k:=k+1 and $\sigma_{k+1} := \sigma_k \circ \{x_k/t_k\}$ and $\Omega_{k+1} := \Omega_k\{x_k/t_k\}$ and go to STEP 2.

Example 4.28. We find a most general unifier for $\Omega = \{P(c, x, f(g(y)), P(z, f(z), f(u))\}$ using the unification algorithm. Then we have:

$\Omega_0 = \Omega, \sigma_0 = \epsilon$
$D_0 = \{c, z\}$

$\sigma_1 = \{z/c\}$
$\Omega_1 = \{P(c, x, f(g(y))), P(c, f(c), f(u))\}$
$D_1 = \{x, f(c)\}$

$\sigma_2 = \{z/c\} \circ \{x/f(c)\} = \{z/c, x/f(c)\}$
$\Omega_2 = \{P(c, f(c), f(g(y))), P(c, f(c), f(u)\}$
$D_2 = \{g(y), u\}$

$\sigma_3 = \sigma_2 \circ \{u/g(y)\} = \{z/c, x/f(c), u/g(y)\}$
$\Omega_3 = \{P(c, f(c), f(g(y)))\}$

$|\Omega_3| = 1$

STOP 1

$\sigma = \sigma_3$

Notation 4.29. Let Ω be a finite and non-empty set of expressions. Then if Ω is unifiable σ_Ω denotes the unifier for Ω obtained by applying the unification algorithm.

Remark 4.30. Notice that the unification algorithm will always stop for any finite nonempty set of expressions. Otherwise, since x_k does not occur in t_k we have that x_k does not occur in Ω_{k+1}, so there would be generated an infinite sequence $\Omega_0, \ldots \Omega_k \ldots$ of finite nonempty sets of expressions with the property that each successive set contains one less variable than its predecessor. This is impossible since Ω contains only a finite number of distinct variables.
As indicated above, if Ω is unifiable, the unification algorithm will always find a most general unifier for Ω. This is proved in the following theorem.

Theorem 4.31. ***Unification Theorem (Robinson):*** *Let Ω be a finite and non empty set of expressions. Then:*

(a) If Ω is unifiable, STOP 1 and σ_Ω is a most general unifier for Ω.

(b) If Ω is not unifiable, STOP 2.

Proof. By remark 4.30 the algorithm always stops.

On one hand, it is clear that STOP1 implies that Ω is unifiable, so if Ω is not unifiable then we arrive to STOP2.

On the other hand STOP2 implies that Ω is not unifiable, so Ω unifiable implies STOP1.

Let us assume Ω is unifiable. We now prove that the unifier σ_Ω is a most general unifier for Ω.

Let θ be a unifier for Ω. We prove the following condition:

(*) For every k, if σ_k is constructed, there exists λ_k such that $\theta = \sigma_k \circ \lambda_k$.

Clearly, (*) implies that σ_Ω is a most general unifier for Ω.

We prove (*) by induction on k. If k=0, then $\sigma_0 = \epsilon$ and $\lambda_0 = \theta$. Let us assume now that k>0 and $\theta = \sigma_k \circ \lambda_k$. If $|\Omega_k| = 1$, σ_{k+1} is not constructed and therefore (*) is satisfied.

Let us assume that $|\Omega_k| > 1$. Then $\Omega_k\lambda_k = (\Omega\sigma_k)\lambda_k = \Omega(\sigma_k\lambda_k) = \Omega\theta$ by the Induction Hypothesis. Now , since θ is a unifier for Ω, we deduce that λ_k is a unifier for Ω_k, and hence λ_k unifies $\{x_k, t_k\}$. Therefore $x_k\lambda_k = t_k\lambda_k$.

So now we have $\sigma_{k+1} = \sigma_k \circ \{x_k/t_k\}$. Let $\lambda_{k+1} = \lambda_k - \{x_k/x_k\lambda_k\}$. Then:

$$\{x_k/t_k\} \circ \lambda_{k+1} = \{x_k/t_k\} \circ (\lambda_k - \{x_k/x_k\lambda_k\}) =$$
$$\{x_k/t_k(\lambda_k - \{x_k/x_k\lambda_k\})\} \cup \{(\lambda_k - \{x_k/x_k\lambda_k\}) =$$
$$\{x_k/t_k\lambda_k\} \cup (\lambda_k - \{x_k/x_k\lambda_k\}) = \lambda_k$$

since x_k does not occur in t_k.

Now $\sigma_{k+1} \circ \lambda_{k+1} = (\sigma_k \circ \{x_k/t_k\}) \circ \lambda_{k+1} = \sigma_k \circ (\{x_k/t_k\} \circ \lambda_{k+1}) = \sigma_k \circ \lambda_k = \theta$ by the Induction Hypothesis. #

Definition 4.32. A substitution $\{x_1/t_1, \ldots, x_n/t_n\}$ is said to be *elementary* when $t_1, \ldots, t_n$ are variables.

Definition 4.33. If $L = \{\phi\}$ and $M = \{\sim \phi\}$ for some literal ϕ, we say L and M are *unitary complementary sets.*

As in propositional logic, given a literal ϕ, we define $\sim \phi = \neg\phi$ if ϕ is an atom, and $\sim \phi = \psi$ if $\phi = \neg\psi$.

Definition 4.34. Let ϕ_1, ϕ_2, ϕ be clauses. ϕ is said to be a *resolvent* of ϕ_1, ϕ_2 if:

1. There exist elementary substitutions ζ, γ such that $\phi_1\zeta$ and $\phi_2\gamma$ are clauses with no variables in common.

2. There exist $L \subseteq Mb(\phi_1), M \subseteq Mb(\phi_2)$ with $L, M \neq \emptyset$ such that the set of atoms N in $L\zeta \cup M\gamma$ is unifiable.

3. $L\zeta\sigma_N$ and $M\gamma\sigma_N$ are complementary unitary sets.

4. $Mb(\phi) = (Mb(\phi_1) - L)\zeta\sigma_N \cup (Mb(\phi_2) - M)\gamma\sigma_N$

Example 4.35. Let $\phi_1 = P(x) \vee Q(f(x)) \vee Q(z)$ and
$\phi_2 = R(x, y) \vee \neg Q(x)$.

We take $\zeta = \{x/u\}, \gamma = \epsilon$, and $L = \{Q(f(u)), Q(z)\}, M = \{\neg Q(x)\}$.

$N = \{Q(f(u)), Q(z), Q(x)\}$ is unifiable by $\{z/f(u), x/f(u)\}$, therefore $P(u) \vee R(f(u), y)$ is a resolvent of ϕ_1, ϕ_2.

Definition 4.36. Let Φ be a finite set of clauses. Let $H^* = H^*(\Phi)$ and $P \subseteq H^*$. Let ϕ be such that $\phi \in \Phi$. A *P-instance of* ϕ is a formula obtained by replacing the variables in ϕ by members of P.

Our aim now is to generalize the Basic Resolution Theorem to the context of the first order logic. For this, we need to prove the following Lemma.

Lemma 4.37. ***(Lifting Lemma)*** *Let Φ be a finite set of clauses, $H^* = H^*(\Phi)$, and $P \subseteq H^*$. Let $\phi_1, \phi_2 \in \Phi$. Let ϕ_1', ϕ_2' be P-instances of ϕ_1, ϕ_2 respectively. Let ϕ' be a resolvent of ϕ_1' and ϕ_2'. Then there exists a resolvent ϕ of ϕ_1, ϕ_2 such that ϕ' is a P-instance of ϕ.*

Proof. Let $z_1, \ldots, z_n$ be the variables occurring in ϕ_1, and $u_1, \ldots, u_l$ the variables occurring in ϕ_2.

As ϕ_1' is a P-instance of ϕ_1, there exists $\alpha = \{z_1/t_1, \ldots, z_n/t_n\}$ such that $t_1, \ldots, t_n \in P$ and $\phi_1' = \phi_1\alpha$.

Since ϕ_2' is a P-instance of ϕ_2, there exists $\beta = \{u_1/s_1, \ldots, u_l/s_l\}$ such that $s_1, \ldots, s_l \in P$ and $\phi_2' = \phi_2\beta$.

As ϕ' is a resolvent of ϕ_1', ϕ_2', there exist $L \subseteq Mb(\phi_1), M \subseteq Mb(\phi_2)$ such that the following conditions are satisfied:

(1) $L\alpha$ and $M\beta$ are complementary unitary sets.

(2) $Mb(\phi') = (Mb(\phi_1) - L)\alpha \cup (Mb(\phi_2) - M)\beta$

Now let $\zeta = \{z_1/x_1, \ldots, z_n/x_n\}, \gamma = \{u_1/y_1, \ldots, u_l/y_l\}$ where $x_1, \ldots, x_n, y_1, \ldots, y_l$ are new variables. Let $\theta = \{x_1/t_1, \ldots, x_n/t_n, y_1/s_1, \ldots, y_l/s_l\}$.

Then $\phi_1\zeta\theta = \phi_1\alpha,\ \phi_2\gamma\theta = \phi_2\beta$, and consequently $L\zeta\theta = L\alpha,\ M\gamma\theta = M\beta$.

Let N be the set of atoms in $L\zeta \cup M\gamma$. Since $L\zeta\theta, M\gamma\theta$ are complementary unitary sets and N is the set of atoms in $L\zeta \cup M\gamma$ we have that θ is a unifier for N, so clearly N is unifiable.

Since σ_N is a most general unifier, there exists a substitution λ such that $\theta = \sigma_N \circ \lambda$.

Let ϕ be such that $Mb(\phi) = (Mb(\phi_1) - L)\zeta\sigma_N \cup (Mb(\phi_2) - M)\gamma\sigma_N$. Then ϕ is a resolvent of ϕ_1, ϕ_2 because σ_N is a unifier for N, and we also have that

$$Mb(\phi\lambda) = (Mb(\phi_1) - L)\zeta\sigma_N\lambda \cup (Mb(\phi_2) - M)\gamma\sigma_N\lambda =$$

$$= (Mb(\phi_1) - L)\zeta\theta \cup (Mb(\phi_2) - M)\gamma\theta =$$

$$= (Mb(\phi_1) - L)\alpha \cup (Mb(\phi_2) - M)\beta = Mb(\phi')$$

Therefore ϕ' is a P-instance of ϕ. #

In the sequel we will proceed as in propositional logic.

Definition 4.38. If Φ is a finite set of clauses, we define $R(\Phi) = \Phi \cup \{\phi$: there exist $\phi_1, \phi_2 \in \Phi$ such that ϕ is a resolvent of $\phi_1, \phi_2\}$.

We define also $R^0(\Phi) = \Phi$, $R^{n+1}(\Phi) = R(R^n(\Phi))$.

In the same manner as in propositional logic, the concept of proof by resolution is defined and it is proved that for a finite set $\Phi \cup \{\phi\}$ of clauses

$$\Phi \underset{R}{\vdash} \phi \text{ iff } \phi \in R^n(\Phi) \text{ for some n} \geq 0$$

.

Lemma 4.39. *Let $H^* = H^*(\Phi)$ and $P \subseteq H^*$. Then $R^n(P(\Phi)) \subseteq P(R^n(\Phi))$ for every n⩾0, where $P(\Phi)$ is the set of instances of formulas in Φ with terms in P.*

Proof. By induction on n. For the case n=0 the proof is immediate.

- Assume that n=1. Let $\phi' \in R(P(\Phi))$. Then we distinguish the following two cases:

  Case 1. $\phi' \in P(\Phi)$.

  $\Phi \subseteq R(\Phi)$ implies that $\phi' \in P(R(\Phi))$.

  Case 2. $\phi' \notin P(\Phi)$.

  There exist $\phi'_1, \phi'_2 \in P(\Phi)$ such that ϕ' is a resolvent of ϕ'_1, ϕ'_2. The Lifting Lemma asserts that there exists $\phi \in R(\Phi)$ such that ϕ' is a P-instance of ϕ, therefore $\phi' \in P(R(\Phi))$.

- Finally, assume that n≥2. Let us suppose $R^{n-1}(P(\Phi)) \subseteq P(R^{n-1}(\Phi))$. If $\phi' \in R^n(P(\Phi))$ then $\phi' \in R(R^{n-1}((P(\Phi)))$ and by Induction Hypothesis we have that $\phi' \in R(P(R^{n-1}(\Phi)))$ therefore by case n=1 $\phi' \in P(R^n(\Phi))$.

#

Just as was done for propositional logic, the correctness and completeness of resolution calculus are shown by the following theorem:

Theorem 4.40. ***Resolution Theorem:*** *Let Φ be a finite set of clauses in the first order logic. Then*

$$Uns\ \alpha_\Phi \text{ iff } \square \in R^n(\Phi) \text{ for some } n \geqslant 0$$

.

Proof. $\Leftarrow$) Let us suppose that $\square \in R^n(\Phi)$. Therefore there exists a proof by resolution of $\square$ from Φ, $(\phi_1, \ldots, \phi_k, \square)$. And let us suppose α_Φ has a model I. Let $\overline{x_i}$ be the variables occurring in ϕ_i,(i=1,...,k).

As $I \models \alpha_\Phi$, we have $I \models \forall\overline{x_i}\ \phi_i$ for every i, therefore I$\models \square$, which is a impossible.

$\Rightarrow$) Let H^* be the Herbrand Universe of Φ. By Herbrand's Theorem $Uns\ \alpha_\Phi$ implies that there exists a finite $P \subseteq H^*$ such that $Uns^\circ P(\Phi)$, and by the Basic Resolution Theorem it follows that there exist a finite $P \subseteq H^*$ and an n$\geqslant$0 such that $\square \in R^n(P(\Phi))$. Then Lemma 4.39 would yield $\square \in P(R^n(\Phi))$ and consequently $\square \in R^n(\Phi)$.

#

Now some examples of how the resolution method applies are given. First we shall see that Resolution is much more efficient than Herbrand's method:

Example 4.41. Let $\Phi = \{\phi_1 = P(x, g(x), y, h(x, y), z, k(x, y, z)),$
$\phi_2 = \neg P(u, v, l(v), w, f(v, w), x)\}$. We can rename the variable x in ϕ_2 by x' and we can take

$$L = \{P(x, g(x), y, h(x, y), z, k(x, y, z))\}$$

and

$$M = \{\neg P(u, v, l(v), w, f(v, w), x')\}$$

Now we consider the set

$$N = \{P(x, g(x), y, h(x, y), z, k(x, y, z)), P(u, v, l(v), w, f(v, w), x')\}$$

which is unifiable by

$$\sigma_N = \{x/u, v/g(u), y/l(g(u)), w/h(u, l(g(u)), z/f(g(u), h(u, l(g(u))),$$
$$x'/k(u, l(g(u)), f(g(u), h(u, l(g(u))))\}$$

Thus σ_N unifies N, so $\square \in R(\Phi)$ and consequently Uns α_Φ. If we had applied Herbrand's method, we would have that the first i such that $Uns^\circ H_i(\Phi)$ is 5, so $|H_5| = 10^{64}, |H_5(\Phi)| \approx 10^{256}$.

Example 4.42. Now it is shown by resolution that if and associative system S is such that all the equations of the form of $x \circ a = b$, $a \circ y = b$ have a solution, then S has a right-neutral element.

By P(x,y,z) we express that $x \circ y = z$. Then the premises can be expressed as:

$$\phi_1 = \forall x \forall y \forall z \forall u \forall v \forall w((P(x,y,u) \wedge P(y,z,v) \wedge P(x,v,w)) \rightarrow P(u,z,w))$$

$$\phi_2 = \forall x \forall y \forall z \forall u \forall v \forall w((P(x,y,u) \wedge P(y,z,v) \wedge P(u,z,w)) \rightarrow P(x,v,w))$$

$$\phi_3 = \forall x \forall y \exists z(P(z,x,y))$$

$$\phi_4 = \forall x \forall y \exists z(P(x,z,y))$$

where ϕ_1 and ϕ_2 formalize the associativity and ϕ_3 and ϕ_4 formalize the fact that all equations in the form of $x \circ a = b$, $a \circ y = b$ have a solution. The existence of a right neutral element is expressed by

$$\phi = \exists x \forall y(P(y,x,y))$$

So now we would like to prove by resolution that $\{\phi_1, \phi_2, \phi_3, \phi_4\} \models \phi$. Converting $\phi_1, \phi_2, \phi_3, \phi_4$ and $\neg\phi = \forall x \exists y \neg P(y,x,y)$ into S.S.F. we have:

1. $\neg P(x,y,u) \vee \neg P(y,z,v) \vee \neg P(x,v,w) \vee P(u,z,w)$ Premise
2. $\neg P(x,y,u) \vee \neg P(y,z,v) \vee \neg P(u,z,w) \vee P(x,v,w)$ Premise
3. P(f(x,y),x,y) Premise
4. P(x,g(x,y),y) Premise
5. ¬P(h(x),x,h(x)) Premise
6. ¬P(x',z,x')∨P(y',z,y') (1,3)
   L=$\{\neg P(x,y,u), \neg P(x,v,w)\}$
   M=$\{P(f(x,y),x,y)\}$
   N=$\{P(x,y,u), P(x,v,w), P(f(x',y'),x',y')\}$
   $\sigma_N = \{x/f(x',y'), y/x', v/x', u/y', w/y'\}$

7. ¬P(x',x,x') (5,6)

   L=$\{\neg P(h(x), x, h(x))\}$

   M=$\{P(y', z, y')\}$

   N=$\{P(h(x), x, h(x)), P(y', z, y')\}$

   $\sigma_N = \{y'/h(x), z/x\}$

8. $\square$ (4,7)

   L=$\{P(x, g(x, y), y)\}$

   M=$\{\neg P(x', x, x')\}$ now, renaming x as z

   N=$\{P(x, g(x, y), y), P(x', z, x')\}$

   $\sigma_N = \{x/x', z/g(x', y), y/x'\}$

Example 4.43. We now consider an example from group theory. We represent the group operation by $\circ$. Again P(x,y,z) means that $x \circ y = z$. Then the axioms of group theory can be expressed by the following formulas:

1) $\forall x \forall y \exists z P(x, y, z)$ (closure under $\circ$)

2) $\forall x \forall y \forall z \forall u \forall v \forall w((P(x, y, u) \wedge P(y, z, v) \rightarrow (P(x, v, w) \leftrightarrow P(u, z, w)))$ (associativity)

3) $\exists x(\forall y P(x, y, y) \wedge \forall y \exists z P(z, y, x))$ (existence of a left-neutral element and existence of left-inverses)

   Now we want to prove that the existence of right-inverses follows from 1),2) and 3). The existence of right inverses is expressed by the formula:

4) $\exists x(\forall y P(x, y, y) \wedge \forall y \exists z P(y, z, x))$

Converting 1), 2), 3) and the negation of 4) into clause form yields formulas a)-f), where, m $(2 - ary)$, e $(0 - ary)$, i $(1 - ary)$ and k $(1 - ary)$ are newly introduced Skolem functions. A resolution refutation from (a)-(f), and therefore, a proof of unsatisfiability is given now:

(a) $P(x, y, m(x, y))$ Premise

(b) $\neg P(x, y, u) \vee \neg P(y, z, v) \vee \neg P(x, v, w) \vee P(u, z, w)$ Premise

(c) $\neg P(x, y, u) \vee \neg P(y, z, v) \vee \neg P(u, z, w) \vee P(x, v, w)$ Premise

(d) $P(e, y, y)$ Premise

(e) $P(i(y), y, e)$ Premise

(f) $\neg P(x, j(x), j(x)) \vee \neg P(k(x), z, x)$ Premise

(g) $\neg P(k(e), z, e)$ a resolvent of (f) and (d)

$L = \{P(e, y, y)\}$

$M = \{\neg P(x, j(x), j(x))\}$

$N = \{P(e, y, y), P(x, j(x), j(x))\}$

$\sigma_N = \{x/e, y/j(e)\}$

(h) $\neg P(x, y, k(e)) \vee \neg P(y, z, v) \vee \neg P(x, v, e)$ a resolvent of (g) and (b)

$L = \{P(u, z, w)\}$

$M = \{\neg P(k(e), z, e)\}$

$N = \{P(u, z, w), P(k(e), z, e)\}$

$\sigma_N = \{u/k(e), w/e\}$

(i) $\neg P(i(v), w, k(e)) \vee \neg P(w, z, v)$ a resolvent of (h) and (e), renaming the variable y in (h) by w

$L = \{P(i(y), y, e)\}$

$M = \{\neg P(x, v, e)\}$

$N = \{P(i(y), y, e), P(x, v, e)\}$

$\sigma_N = \{x/i(v), y/v\}$

(j) $\neg P(i(v), e, k(e))$ a resolvent of (i) and (d)

$L = \{P(e, y, y)\}$

$M = \{\neg P(w, z, v)\}$

$N = \{P(e, y, y), P(w, z, v)\}$

$\sigma_N = \{w/e, z/y, v/y\}$

(k) $\neg P(i(t), y, u) \vee \neg P(y, z, e) \vee \neg P(u, z, k(e))$ a resolvent of (j) and (c), renaming the variable v in (j) as t

$L = \{P(x, v, w)\}$

$M = \{\neg P(i(v), e, k(e))\}$

$N = \{P(x, v, w), P(i(t), e, k(e))\}$

$\sigma_N = \{x/i(t), v/e, w/k(e)\}$

(l) $\neg P(i(t), y, e) \vee \neg P(y, k(e), e)$ a resolvent of (k) and (d), renaming the variable y in (d) by r

$L = \{P(e, y, y)\}$

$M = \{\neg P(u, z, k(e))\}$

$N = \{P(e, r, r), P(u, z, k(e))\}$

$\sigma_N = \{u/e, z/k(e), r/k(e)\}$

(m) $\neg P(i(t), i(k(e)), e)$ a resolvent of (l) and (e), renaming the variable y in (e) by s

$L = \{P(i(y), y, e)\}$

$M = \{\neg P(y, k(e), e)\}$

$N = \{P(i(s), s, e), P(y, k(e), e)\}$

$\sigma_N = \{s/k(e), y/i(k(e))\}$

(n) $\square$ a resolvent of (m) and (e)

$L = \{\neg P(i(t), i(k(e)), e)\}$

$M = \{P(i(y), y, e)\}$

$N = \{P(i(t), i(k(e)), e), P(i(y), y, e)\}$

$\sigma_N = \{t/i(k(e)), y/i(k(e))\}$

5 Logic Programming

In this section Logic Programming is introduced, showing that the execution of a logic program can be understood as the automated deduction of the empty clause from an unsatisfiable set of clauses by using resolution.

In artificial intelligence, logic programs are broadly used to program expert systems in order to emulate the decision-making ability of a human expert, and to solve practical problems in the NP class. An example of the latter could be the coloring of a map, using only four different colors, in such a way that no two adjacent regions share the same color. This can always be done as stated by the four color map theorem, that was finally proved in 1976 after being an open problem for over a century, and was the first major theorem to be proved using a computer. To find a correct coloring, a program in Prolog language is given in the final section as an example.

5.1 Computational model of Logic Programs

In order to describe the computational model of Logic Programs some notions must be previously introduced.

Definition 5.1. *Identifiers* are finite strings formed by letters, digits and underscores beginning with a letter or an underscore.

Definition 5.2. A *logic variable* is an identifier beginning with an upper-case letter or an underscore.

Definition 5.3. An *atom* is either an identifier that begins with a lower-case letter or a finite string between single quotes, and a *constant* is either a number or an atom. In a logic program, all constant symbols, function symbols and all predicate symbols are atoms.

Definition 5.4. A *fact* is an atomic formula in some first order language, and a *rule* is a formula in some first order language in the form:

$$(\phi_1 \wedge \ldots \wedge \phi_n) \rightarrow \phi$$

where $n \geq 1$ and $\phi_1, \ldots, \phi_n, \phi$ are atomic formulas.

Remarks 5.5.

1. Note that since $(\phi_1 \wedge \ldots \wedge \phi_n) \rightarrow \phi \equiv \neg\phi_1 \vee \ldots \vee \neg\phi_n \vee \phi$ we have that every rule is a clause.
2. $\wedge$ is denoted by ','
3. The implication symbol is denoted by ':-'
4. A rule $(\phi_1 \wedge \ldots \wedge \phi_n) \rightarrow \phi$ is denoted by ϕ :- $\phi_1, \ldots, \phi_n$. ϕ is called the *head* of the rule and $\phi_1, \ldots, \phi_n$ is called the *body*.

Definition 5.6. A *logic program* is a finite set of facts and rules.

Remark 5.7. If ϕ is a formula of a logic program and $x_1, \ldots, x_n$ are the variables that occur in ϕ, the meaning of ϕ is $\forall x_1 \ldots \forall x_n \phi$.

Remark 5.8. Given two formulas ϕ and ψ and a variable x that occurs in ϕ but not in ψ. Then:

$$\forall x(\phi \rightarrow \psi) \equiv \forall x(\neg\phi \vee \psi) \equiv \forall x\neg\phi \vee \psi \equiv \neg\exists x\phi \vee \psi \equiv \exists x\phi \rightarrow \psi$$

Definition 5.9. A *goal* for a logic program is a first-order formula $\phi_1, \ldots, \phi_n$ where $n \geq 1$ and $\phi_1, \ldots, \phi_n$ are atomic formulas.

If ϕ is a goal and $x_1, \ldots, x_k$ are the variables that occur in ϕ, the meaning of ϕ is $\exists x_1 \ldots \exists x_k \phi$.

Notation 5.10. We will denote by $\overline{x}$ a finite sequence of variables.

Remark 5.11. If $\{\phi_1(\overline{x_1}), \ldots, \phi_n(\overline{x_n})\}$ is a logic program and $\phi(\overline{z})$ is a goal, the interpreter tries to show that $\{\forall\overline{x_1}\phi_1, \ldots, \forall\overline{x_n}\phi_n\} \models \exists\overline{z}\phi$.

Now the abstract interpreter for logic programs (also called computational model of logic programs) will be introduced.

Abstract Interpreter

Inputs: A logic program P and a goal X.

1. Put k=0, $R_k = \neg X$.
2. While ($R_k \neq \square$) do
   a) Chose the first literal $\neg\phi$ in R_k and a renamed clause $\phi_k = (\phi' :- \phi'_1, \ldots, \phi'_n)$ from P such that $\{\phi, \phi'\}$ is unifiable. If no such literal and clause exist, go to STEP 3. Otherwise continue.
   b) R_{k+1} = resolvent of R_k, ϕ_k with respect to $\{\phi, \phi'\}$.
   c) k=k+1.
3. If $R_k = \square$ then output='true', otherwise output='failure'.

In each step the interpreter generates a formula R_k in the form of $\neg\alpha_1 \vee \ldots \vee \neg\alpha_m$.

In order to avoid the negation symbols preceding each atom in R_k, it is more efficient to work with the negation of the formula in each step, this is, $\alpha_1, \ldots, \alpha_m$. By doing this an equivalent but more efficient algorithm is obtained. We denote by $\blacksquare$ the empty conjunction. Hence $\blacksquare$ is a tautology.

Inputs: A logic program P and a goal X.

1. Put k=0, $O_k = X$.
2. While ($O_k \neq \blacksquare$) do
   a) Chose the first literal ϕ in O_k and a renamed clause $\phi_k = (\phi' :- \phi'_1, \ldots, \phi'_n)$ from P such that $\{\phi, \phi'\}$ is unifiable. If no such literal and clause exist, go to STEP 3. Otherwise continue.
   b) O_{k+1} is the formula obtained by replacing in O_k the formula ϕ by $\phi'_1, \ldots, \phi'_n$ and then applying the substitution $\sigma_{\{\phi,\phi'\}}$.
   c) k=k+1.
3. If $O_k = \blacksquare$ then output='true', otherwise output='failure'.

Remark 5.12. At each step k, if $O_k = \phi_1, \ldots, \phi_n$ then $R_k = \neg\phi_1 \vee \ldots \vee \neg\phi_n$. So we have that $\neg O_k \equiv R_k$.

Definition 5.13. A *trace of computation* of a logic program is a sequence of goals generated by the interpreter, $O_0, O_1, \ldots$

Notation 5.14. In order to write out a trace of computation, an arrow is written between the goal at step k and the goal at step k+1. Matches to be carried out (variables unified by the interpreter) are specified at each step to make the process clear.

O_1

matches to be carried out

$\longrightarrow$

O_2

matches to be carried out

$\longrightarrow$

O_3

$\vdots$

5.2 Prolog interpreter

Mainly two decisions must be taken in order to convert the abstract interpreter for logic programs into a suitable form for any concrete programming language. In first place, the arbitrary choice of which goal in the resolvent to reduce. Second, the non-deterministic choice of the clause from the program to effect the reduction must be implemented.

The problem remains in the fact that, in general, there are many possibilities to find two resolvable clauses for producing new resolvents. Among the possible resolution steps only a few might lead to the derivation of the empty clause. A possibility to improve the efficiency of the general resolution algorithm is a refinement of resolution called *SLD-resolution*, which stands for linear resolution with selection function for definite clauses. SLD-resolution is an input resolution, meaning one of the two parent clauses from which each resolvent is derived is a clause in the program, and it is also linear, meaning the other parent clause is the previous resolvent. In a parallel manner as in which completeness of resolution was proved, it is shown that SLD-resolution is complete for Horn clauses, on which is based the Prolog interpreter.

The Prolog interpreter is obtained from the abstract interpreter by replacing the non-deterministic choice of a clause in the program by sequential search for a unifiable clause and backtracking.

This search occurs in the following manner:

At step k Prolog interpreter chooses the first atom ϕ in O_k and the first clause in the program whose head unifies with ϕ. In the case such clause does not exist, Prolog interpreter goes back to O_{k-1}, and then chooses the first atom ϕ in O_{k-1} and the next clause in the program whose head unifies with ϕ.

Also this backtracking process is carried out by Prolog interpreter whenever it obtains a solution for the goal. In order to obtain further solutions the interpreter goes back to the previous goal O_{k-1}, and then again chooses the first atom ϕ in O_{k-1} and the next resolvent in the program whose head unifies with ϕ.

5.3 Basic Built-in predicates in Prolog

1) Prolog uses the following notation for basic arithmetical operators:

   (a) + addition
   (b) - subtraction
   (c) * multiplication
   (d) / division
   (e) // integer division
   (f) mod remainder

2) The arithmetical predicates:

   (a) =:=
   (b) =/= (inequality)
   (c) =<
   (d) >=
   (e) >

   These predicates and operators are executed by Prolog's interpreter directly, this means without using resolution.

3) =.

   It's meaning is unification. If e_1, e_2 are expressions, $e_1 = e_2$ means that e_1 and e_2 unify. With respect to unification, if a variable in some of the expressions is instantiated to a constant value c, then the variable is replaced by its value c. Thus, when a variable is instantiated, what counts for unification is the value of the variable.

4) The predicate is.

   It has the form V is E, where V is a variable and E is an expression. If V is not instantiated, the value of the expression E is worked out and assigned to V. If V is instantiated, its value must coincide with the value of E.

5) The predicate read.

   It has the form read(X), where X is a variable. Reads the next Prolog term from the current input stream and unifies it with the argument taken.

6) The predicate write.

   It has the form write (t), where t is a term. Writes the term taken as argument to the current output, using brackets and operators where appropriate.

Remark 5.15. In the Prolog language single underscore (_) denotes an anonymous variable and means "any term".

Example 5.16. Here is an example of the trace of computation of a program in Prolog Language that finds the greatest common divisor of two given natural numbers using the Euclidean algorithm. This is done by means of a single predicate gcd(A,B,C), which means the greatest common divisor of A and B is C, and is defined recursively as follows:

```
gcd(X,0,X).
gcd(X,Y,Z) :- U is X mod Y , gcd(Y,U,Z).
```

So, in order to find the greatest common divisor of 60 and 42, for example, the goal to be set would be gcd(60,42,A). Here is the trace of computation of the program:

gcd(60,42,A)

X=60, Y=42, Z=A

$\longrightarrow$

gcd(42,18,A)

X=42, Y=18, Z=A

$\longrightarrow$

gcd(18,6,A)

X=18, Y=6, Z=A

$\longrightarrow$

gcd(6,0,A)

A=6

$\longrightarrow$

■

5.4 Lists

Lists are the only data structure in Prolog. A *list* is an expression of the form $[t_1, \ldots, t_n]$, where $t_1, \ldots, t_n$ are first order terms.

The empty list is denoted by [ ], and a non-empty list is denoted by [X|R], where X is the first element of the list and R is the list formed by the rest of the elements.

With respect to unification, the interpreter considers a list [X|R] as a function of two arguments: X and R.

Basic Built-in predicates for lists

(1) member(X,Y) Is true if X is a member of the list Y

(2) append(L1,L2,L3) True if L3 is the concatenation of lists L1 and L2

(3) select(X,L1,L2) True when L2 is the result from removing X from list L1

(4) reverse(L1,L2) True when elements of L2 are in reverse order compared to L1

(5) permutation(L1,L2) True when L2 is a permutation of L1

(6) sort(L1,L2) True if L2 is an ordered list containing the elements of L1. Duplicates are removed.

(7) sublist (L1,L2) True if L1 is a sublist of L2.

(8) length(L,C) True if the length of L is C

(9) last(L,X) True if the last element of L is X

(10) findall(R, ...goal... ,L) Includes in the list L all the elements R that satisfy the goal.

5.5 The negation predicate

The Prolog language uses an extension of the computational model for logic programs earlier described. In this extension, the negation predicate is used for atomic formulas in such a way that if ϕ is an atomic formula then not(ϕ) fails if there is some computation for ϕ that succeeds, and not(ϕ) succeeds if every computation for ϕ stops on fail.

5.6 Examples of Logic Programs in the Prolog language

The purpose of this section is to show how we can find a solution by means of the Prolog language for various practical problems which belong to the NP class.

Prolog is particularly useful for solving problems in the NP class, because of the backtracking process carried out by the interpreter.

These examples have been run in the SWI Prolog.

Example 5.17. A salesman has to visit a number of cities minimizing the total traveling distance. The salesman is able to choose where to start as well as the order in which to visit the cities. For this, the salesman is provided with a list of all cities to be visited as well as a list of the minimum distances between every pair of cities.

The program consists of a data base including the distances between every pair of cities. These are expressed using the predicate dist(X,Y,D) meaning that the distance between cities X and Y is D.

The predicate write_out is used to write out all the elements in a given list, and tour(X,L), which is defined by means of the built-in predicate permutation(X,L), is satisfied when X is a tour that visits all cities in the list L once and only once, as list X will be a permutation of elements in list L.

The predicate dist_tot(L,D), defined recursively, is true when the total distance of the tour L is D kilometers, and is used in the predicate tour_dist(T,L,D) which means that T is a tour for the list of cities L, and its total distance is D.

The built-in predicate least(X,L) means X is the element of least value in the list L. The predicate tour_min(L) will yield the minimum tour for the cities in the list L by using the former predicates along with the predefined predicates member and findall.

```
dist(bcn,gir,103).
dist(bcn,tar,83).
dist(bcn,lle,162).
dist(bcn,man,48).
dist(bcn,vil,59).
dist(bcn,olo,92).
dist(gir,tar,162).
dist(gir,lle,187).
dist(gir,man,121).
dist(gir,vil,117).
dist(tar,lle,76).
dist(tar,man,83).
dist(tar,vil,94).
dist(lle,man,100).
dist(lle,vil,94).
dist(man,vil,43).
dist(olo,man,133).
dist(olo,lle,167).
dist(olo,tar,157).
dist(olo,gir,35).
dist(olo,bcn,92).
dist(ter,bcn,24).
dist(ter,gir,81).
dist(ter,tar,80).
dist(ter,lle,115).
dist(ter,man,24).
dist(ter,vil,35).
dist(ter,olo,79).


cities(c,[bcn,gir,tar,lle,man,vil,olo,ter]).


write_out([ ]).
write_out([X|L]) :- write(X), nl, write_out(L).


tour(X,L) :- permutation(X,L).


dist_tot([_],0).
dist_tot([X,Y|L],T) :- dist(X,Y,D) , dist_tot([Y|L],U) , T is D+U.


tour_dist(R,L,D) :- tour(R,L) , dist_tot(R,D).


least(_,[ ]).
least(X,[Y|L]) :- least(X,L) , X=<Y.
least(Y,[Y|L]) :- least(Z,L) , Y=<Z.
```

```
tour_min(C) :- cities(C,L) , findall(D,tour_dist(_,L,D),LD) ,
    member(M,LD) , least(M,LD) , tour_dist(X,L,M) , write_out(X).
```

Finally, in order to retrieve the answer from the program, the goal to be set would be would be tour_min(c).

Example 5.18. The county council wishes to assign a radio frequency to each village or city in the county. In order to avoid interference in the radio signals, the demands are that if two villages are less than 20 kilometers away they must be assigned different radio frequencies. The information provided is a list of all distances between every pair of villages.

A list of cities under the name of c, along with a number of frequencies and the distance between every pair of cities in the list are the data base of the program.

The predicate compatible([CX,FX],[CY,FY]), where [CX,FX] and [CY,FY] are pairs of a city and the frequency chosen for it, expresses the assignment of frequencies to be correct. By addi_correct([CX,FX],L), the fact that the pair city/frequency [CX,FX] is compatible with the list of assignments L is expressed. The predicate assign_valid(L) means L is a list of pairs city/frequency that corresponds to a correct assignment, and assign_frequencies(C,L) expresses that L is a valid assignment of frequencies for the cities in the list C. Finally, assignment(Ciu) will cause a valid assignment for the cities in the list under the name of Ciu to be carried out.

```
cities(c,['Barcelona','Sta Coloma','Mataro', 'Sant Cugat','Granollers',
    'Martorell','Monistrol']).
city(X) :- cities(c,L) , member(X,L).
frequencies(f,[1,2,3,4,5]).
frequency(X) :- frequencies(f,L) , member(X,L).

distance('Barcelona','Sta Coloma',5).
distance('Barcelona','Mataro',24).
distance('Mataro','Sta Coloma',19).
distance('Barcelona','Sant cugat',13).
distance('Mataro','Sant cugat',31).
distance('Sta Coloma','Sant cugat',11).
distance('Granollers','Barcelona',24).
distance('Granollers','Sta Coloma',19).
distance('Granollers','Mataro',15).
```

```
distance('Granollers','Sant Cugat',23).
distance('Granollers','Sta Coloma',19).
distance('Martorell','Sant Cugat',13).
distance('Martorell','Barcelona',23).
distance('Martorell','Sta Coloma',24).
distance('Martorell','Mataro',43).
distance('Martorell','Granollers',33).
distance('Martorell','Monistrol',16).
distance('Monistrol','Granollers',36).
distance('Monistrol','Sant Cugat',25).
distance('Monistrol','Mataro',50).
distance('Monistrol','Sta Coloma',35).
distance('Monistrol','Barcelona',37).

write_out([ ]).
write_out([X|L]) :- write(X), nl, write_out(L).

dist(X,X,0).
dist(X,Y,D) :- distance(X,Y,D).
dist(X,Y,D) :- distance(Y,X,D).

compatible([CX,FX],[CY,FY]) :- city(CX), frequency(FX),
    city(CY), frequency(FY), dist(CX,CY,D) , D>20.
compatible([CX,FX],[CY,FY]) :- city(CX), frequency(FX),
    city(CY), frequency(FY), FX\=FY.

addi_correct([CX,FX],[ ]) :- city(CX) , frequency(FX).
addi_correct([CX,FX],[[CY,FY]|L]):- city(CX) , frequency(FX) ,
    city(CY) , frequency(FY) , compatible(X,Y) , addi_correct(X,L).

assign_valid([ ]).
assign_valid([X|L]) :- addi_correct(X,L) , assign_valid(L).

assign_frequencies([ ],[ ]).
assign_frequencies([C|LC],[X|L]) :- X=[C,F] , frequency(F) ,
    assign_frequencies(LC,L).

assignment(Ciu) :- cities(Ciu,C) , assign_frequencies(C,AF) ,
    assign_valid(AF) , write_out(AF).
```

The goal to be set in order to retrieve the answer would be assignment(c).

Example 5.19. An academy is open four hours a day, five days a week, and class-hours are numbered from 1 to 20. There are also a number of professors and a number of groups of students. For every group of students we have a list of the hours at which the group takes its lessons, and for every teacher a list of restrictions, meaning by this the class-hours at which the teacher can not teach. Now we are asked to do the schedule, this is to assign a single group to every teacher, and a single teacher to every group without violating the restrictions.

The data base for this program will consist in a list of groups, and a list of professors. Each one of these lists should be understood a list of terms, being each term the pair of a name and a list. In the case of the groups, we would have the name of the group and the list of the class-hours at which the classes for the group are scheduled, and in the case of the professors each pair would consist of the professor's name and the list of class-hours at which he or she is unavailable.

The structure that supports the assignment is a again list (under the name of h) of terms that express the name of a group and the professor assigned to it. The predicate group_correct(group(G,P),Groups,Professors) means the assignment of professor P to group G fits in with the bindings in the list of Groups and Professors, and by means of the predicate fill_schedule([G|Gs],Groups,Professors) a schedule is carried out.

```
groups(gr,[(a,[1,9]),(b,[2,10]),(c,[3,11]),(d,[4,12]),(e,[5,16]),(f,[6,17]),(g,[7,18]),
    (h,[8,13]),(i,[14,19]),(j,[15,20])]).

professors(pr,[prof('Albert Riera',[1,2,3,4]),prof('Antoni Sanchez',[4,5,6,7]),
    prof('Julia Roca',[1,2,5,6]), prof('Vicenç Martinez',[13,14,15,16]),
    prof('Cristina Grau',[17,18,19,20]),prof('Silvia Pont',[3,4,7,8]),
    prof('Julia Cao',[1,2,3,4,9,10,11,12]),prof('Robert Salla',[]),
    prof('Joan Satorra',[1,2,3,4]),prof('Pep Bou',[5,6,7,8])]).

schedule(h,[group(a,A),group(b,B),group(c,C),group(d,D),group(e,E),group(f,F),
    group(g,G),group(h,H),group(i,I),group(j,J)]).

disjoint([ ],_).
disjoint([A|B],C):- not(member(A,C)), disjoint(B,C).

write_out([ ]).
write_out([X|L]) :- write(X), nl, write_out(L).
```

```
group_correct(group(G,P),Groups,Professors) :- member(prof(P,RP),Professors) ,
    member((G,HG),Groups) , disjoint(RP,HG).

fill_schedule([G|Gs],Groups,Professors) :- G = group(X,Prof) ,
    group_correct(group(X,Prof),Groups,Professors) ,
    select(prof(Prof,_),Professors,Profrest) , fill_schedule(Gs,Groups,Profrest).
fill_schedule([ ],Groups,Professors).

generate(Schedule,Groups,Professors) :- schedule(Schedule,H) ,
    groups(Groups,LG) , professors(Professors,LP) , fill_schedule(H,LG,LP) ,
    write_out(H).
```

A full schedule is carried out when we set fill_schedule(h,gr,pr) as the goal.

Example 5.20. The city council wishes to schedule the chemist night shifts in such a manner that every night there is exactly one chemist open. Each chemist provides the council with a list of nights when it is not able to remain open, and the council sets a list of restrictions, being each restriction a pair of dates to which the same chemist can not be assigned.

Terms chemist(Ch,L) express chemist Ch closes on the days included in L, and the restrictions imposed by the council are contained in a list under the name of res. By means of the predicate fill_day and fill_calendar a chemist is assigned to every day in such a manner that all chemists are able to open on the days for which they are chosen.

The predicate calendar_valid is used to verify the council restrictions are not being infringed. Finally, generate_calendar causes a correct calendar to be generated.

```
chemists(far,[chemist(' Albert Fernandez',[1,3]),chemist('Marquiza',[1,2]),
    chemist('Solernou',[4,5]),chemist('Bou',[7,8])]).

restrictions(res,[[1,2],[2,3],[3,4],[4,5],[5,6],[6,7],[7,8]]).

calendar(cal,[day(1,One),day(2,Two),day(3,Three),day(4,Four),day(5,Five),
    day(6,Six),day(7,Seven),day(8,Eight)]).

write_out([ ]).
```

```
write_out([X|L]) :- write(X), nl, write_out(L).

fill_day(day(D,F),Chemists) :- member(chemist(F,L),Chemists) ,
    not(member(D,L)).

fill_calendar([Day|Days],Chemists) :- fill_day(Day,Chemists) ,
    fill_calendar(Days,Chemists).
fill_calendar([ ],Chemists).

calendar_valid(L,[ ]).
calendar_valid(L,[[R1,R2]|Restrictions]) :- member(day(R1,F1),L) ,
    member(day(R2,F2),L) , F1\ =F2 , calendari_valid(L,Restrictions).

generate_calendar(C,Chemists,Restrictions) :-
    fill_calendar(C,Chemists,Restrictions) , calendar_valid(C,Restrictions).
generate(Cal,Ch,Res) :- calendar(Cal,C) , chemists(Ch,F) , restrictions(Res,R) ,
    generate_calendar(C,F,R) , write_out(C).
```

The goal to be set would be generate(cal,far,res).

Example 5.21. The problem of finding a coloring for a given map, using four colors. Use the program to find an adequate coloring for Europe.

The structure supporting the program is a list of regions, understanding each region as a term that includes the name of the region, a variable whose value will be its color, and the list of colors assigned to the region's neighbors. By means of the predicates color_region and color_map the coloring is completed by establishing the goal color(europe,colors1).

```
colors(colors1,[blau,vermell,groc,verd]).
map(europe,[region(espanya,Espanya,[Franca,Portugal]),
    region(franca,Franca,[Espanya,Italia,Suissa,Belgica,Alemanya]),
    region(portugal,Portugal,[Espanya]),
    region(gran_bretanya,Gran_bretanya,[Irlanda]),
    region(irlanda,Irlanda,[Gran_bretanya]),
    region(belgica,Belgica,[Holanda,Franca,Alemanya]),
    region(holanda,Holanda,[Belgica,Alemanya]),
```

```
region(suissa,Suissa,[Franca,Alemanya,Italia,Austria]),
region(alemanya,Alemanya,[Holanda,Belgica,Suissa,Austria,Rep_txeca,
   Polonia,Dinamarca]),
region(dinamarca,Dinamarca,[Alemanya]),
region(noruega,Noruega,[Suecia,Finlandia,Russia]),
region(suecia,Suecia,[Noruega,Finlandia]),
region(finlandia,Finlandia,[Noruega,Suecia,Russia]),
region(polonia,Polonia,[Alemanya,Rep_txeca,Eslovaquia,Ucraina,Bielorrussia,
   Russia,Lituania]),
region(rep_txeca,Rep_txeca,[Alemanya,Austria,Eslovaquia,Polonia]),
region(italia,Italia,[Franca,Suissa,Austria,Eslovenia,Malta]),
region(malta,Malta,[Italia]),
region(austria,Austria,[Rep_txeca,Alemanya,Suissa,Italia,Eslovenia,
   Eslovaquia,Hungria]),
region(eslovaquia,Eslovaquia,[Rep_txeca,Austria,Hungria,Ucraina,Polonia]),
region(hungria,Hungria,[Eslovaquia,Austria,Ucraina,Rumania,Serbia,Eslovenia,
   Croacia,Bosnia]),
region(eslovenia,Eslovenia,[Italia,Austria,Hungria,Croacia]),
region(croacia,Croacia,[Eslovenia,Hungria,Serbia,Bosnia]),
region(bosnia,Bosnia,[Croacia,Serbia,Montenegro]),
region(montenegro,Montenegro,[Bosnia,Serbia,Albania]),
region(albania,Albania,[Montenegro,Macedonia,Serbia,Grecia]),
region(macedonia,Macedonia,[Albania,Serbia,Bulgaria,Grecia]),
region(serbia,Serbia,[Hungria,Croacia,Bosnia,Montenegro,Albania,Macedonia,
   Bulgaria,Rumania]),
region(bulgaria,Bulgaria,[Serbia,Macedonia,Grecia,Rumania]),
region(grecia,Grecia,[Albania,Macedonia,Bulgaria]),
region(rumania,Rumania,[Ucraina,Moldavia,Hungria,Serbia,Bulgaria]),
region(moldavia,Moldavia,[Rumania,Ucraina]),
region(ucraina,Ucraina,[Bielorrussia,Polonia,Eslovaquia,Hungria,Rumania,
   Moldavia,Russia]),
region(bielorrussia,Bielorrussia,[Letonia,Lituania,Russia,Polonia,Ucraina]),
region(lituania,Lituania,[Estonia,Russia,Bielorrussia]),
region(letonia,Letonia,[Estonia,Lituania,Russia]),
```

```
    region(estonia,Estonia,[Letonia,Russia]),
    region(russia,Russia,[Estonia,Letonia,Lituania,Polonia,Bielorrussia,Ucraina])]).

write_map([ ]).
write_map([region(R,C,N)|L]) :- write(R) , tab (1) , write(C) , nl , write_map(L).

members([ ],Ys).
members([X|Xs],Ys) :- member(X,Ys) , members(Xs,Ys).

color_region(region(Name,Color,Neighbours),Colors) :-
    select(Color,Colors,Colors1) , members(Neighbors,Colors1).

color_map([Region|Regions],Colors) :- color_region(Region,Colors) ,
    color_map(Regions,Colors).
color_map([ ],Colors).

color(Name,Col) :- map(Name,Map) , colors(Col,Colors) , color_map(Map,Colors),
    write_map(Map).
```

References

[1] Chin-Liang Chang; Richard C. Lee: *Symbolic Logic and Mechanical Theorem Proving*, Academic Press Inc, 1973.

[2] W.F. Clocksin, C.S. Mellis: *Programming in Prolog*, Springer, 1987.

[3] M.C. Fitting: *First-Order Logic and Automated Theorem Proving, Springer*, 1996.

[4] R. Farré, R. Nieuwenhuis, P. Nivela, A. Oliveras, E. Rodriguez, J. Sierra: *Lógica para informáticos*, Marcombo, 2011.

[5] H.S. Keisler, A. Mostowski, A. Robinson, P. Suppes, A.S. Troelstra: *Handbook of Mathematical Logic*, Barwise 1971.

[6] A. Robinson; A. Voronkov: *Handbook of Automated Reasoning*, MIT Press, 2001.

[7] Uwe Schöning: *Logic for Computer Scientists*, Birkhäuser, 1989.

[8] L.Sterling, E. Saphiro: *The Art of Prolog*, MIT Press, 1994.